UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Marco Antonio Torrez Rojas

Utilização de Algoritmos Genéticos no Projeto de Caixas-S

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de mestre em Ciência da Computação.

Prof. Ricardo Felipe Custódio, Dr.
Orientador

Florianópolis, Julho de 2002

Utilização de Algoritmos Genéticos no Projeto de Caixas-S

Marco Antonio Torrez Rojas

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Fernando Ostuni Gauthier, Dr.

Coordenador do Curso

Banca Examinadora

Prof. Ricardo Felipe Custódio, Dr.

Orientador

Prof. Jeroen Antonius Maria van de Graaf , Dr.

Prof. Paulo Sérgio da Silva Borges , Dr.

”A alegria do triunfo jamais poderia ser experimentada se não existisse a luta, que é a que determina a oportunidade de vencer.- Gonzalez Pecotche.

Ofereço este trabalho aos meus pais, como demonstração de gratidão, carinho e respeito por todos os seus esforços e resignações durante as diversas fases de minha vida.

Agradecimentos

Primeiramente, ao meu orientador, professor Ricardo Felipe Custódio, que com o seu entusiasmo pela ciência, se excedeu em seu papel e me motivou para a realização deste trabalho.

À minha esposa Giselle, que soube me suportar e incentivar durante a realização deste trabalho e em muitos outros desafios de minha vida.

Aos pesquisadores da área que disponibilizam material de suas pesquisas para consulta pública, em especial aos Professores: Amr Mohamed Youssef (Universidade do Cairo, Egito), Kwangjo Kim (Universidade de Comunicação e Informação, Corea), Willian Millan e Ed Dawson (Universidade de Tecnologia de Queensland, Australia), pelo suporte prestado durante este projeto.

Ao Professor Claudio César de Sá (UDESC-CCT), por me introduzir no universo dos Algoritmos Genéticos e pelas orientações durante a realização deste trabalho.

Aos grandes mestres e alunos da turma de 1999 do curso de Mestrado em Automação Industrial (UDESC-CCT), em especial ao aluno e companheiro César Silvestre pelas discussões e trabalhos realizados durante as disciplinas cursadas, mais especificamente as de nossa área de concentração (Inteligência Artificial).

Aos grandes mestres das disciplinas ministradas no curso de Mestrado realizado em Joinville (UDESC) em parceria com a UFSC, que contribuíram com suas experiências de vida e seus conhecimentos para ampliar a minha visão da área e aguçar minha vontade de aprender mais.

Aos amigos de sala Glauco, Mehran, Marco Andre, Kariston, Edino,

Augusto, Caio, Marcos David, e outros, que tornaram o ambiente amistoso e favorável ao aprendizado com a troca de experiências profissionais.

Aos colegas de trabalho do departamento de Ciências da Computação da UDESC.

Ao amigo Charles Christian Miers por todo o apoio e incentivo durante a realização deste projeto de pesquisa e trabalhos realizados.

À empresa LockNet Security Solutions por acreditar no meu trabalho e me apoiar na realização deste projeto de pesquisa.

As minhas famílias de São Paulo e Joinville, meus pais Dr. Justiniano e Dona Cristina e minhas irmãs Dra. Marianela, Rita e Carmen (mesmo que distante muito presente em toda minha vida), meus sogros Sr. João e Dona Eli Ana e cunhado Carlos.

Ao amigo Valmor Adami Jr pelo suporte e auxílio durante o desenvolvimento dos programas utilizados neste trabalho.

À Fundação Softville e seu gestor Sr. Ademir Rossi por permitir a utilização dos laboratórios da instituição para realização dos testes da implementação.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

AES	Advanced Encryption Standard Padrão Avançado de Encriptação
AG	Algoritmos Genéticos
ANSI	American National Standards Institute Instituto Nacional Americano de Padrões
BIC	Bit Independence Criterion Critério de Independência do Bit de Saída
DES	Data Encryption Standard Padrão de Encriptação de Dados
FIPS	Federal Information Processing Standards Padrão Federal de Processamento da Informação
GF	Galois Field Corpo de Galois
IA	Inteligência Artificial
IBM	International Business Machines Corporation Corporação Internacional de Máquinas de Negócio
IDEA	International Data Encryption Algorithm Algoritmo Internacional de Encriptação de Dados
ISO	International Organization for Standardization Organização Internacional de Padronização
MIT	Massachussets Institute of Tecnology Instituto de Tecnologia de Massachusetts
NBS	National Bureau of Standards Agência Nacional de Padrões
NIST	National Institute of Standards and Technology Instituto Nacional de Padrões e Tecnologia
NSA	National Security Agency Agência Nacional de Segurança
SAC	Strict Avalanche Criterion Critério Rigoroso de Avalanche
SDES	Simplified Data Encryption Standard Padrão Simplificado de Encriptação de Dados
SPN	Substitution-Permutation Network Rede de Substituição-Permutação

Resumo

Uma parcela considerável dos algoritmos de criptografia simétrica utiliza uma estrutura de substituição S ou simplesmente Caixa-S para prover a não linearidade da cifra. A não linearidade é um dos requisitos para garantir a segurança do algoritmo contra ataques de criptoanálise. Várias técnicas têm sido utilizadas e reportadas na literatura para o projeto de uma boa Caixa-S. Neste trabalho, são utilizados algoritmos genéticos - AG no projeto destas Caixa-S. Foram projetadas novas Caixas-S para o cifrador Padrão de Criptografia Avançado - AES, e avaliadas através da medida de sua não linearidade. Estas novas Caixas-S foram comparadas com as Caixas-S originais do cifrador AES.

Palavras chaves: Caixas-S, Algoritmos Genéticos; Cifradores de Bloco Simétricos; Criptografia.

Abstract

A considerable part of symmetric algorithms of cryptography uses a structure of substitution S or simply S-Box in order to provide the non linearity of the cypher. The non linearity is one of the requirements to assure the security of the algorithm against attacks of cryptanalysis. Several techniques have been used and reported in the literature for the design of a good S-Box. In this paper genetic algorithms are used in the design of these S-Boxes. New S-Boxes were designed for the cypher Advanced Cryptography Standard - AES, and evaluated through the measurement of their non linearity. These new S-Boxes were compared to the originals S-Boxes of the cypher AES.

Key Words: S-Box, Genetic Algoritms; Block Ciphers; Cryptrography.

Capítulo 1

Introdução

Com a forte adoção da tecnologia de Redes de Computadores, obtivemos grandes ganhos e facilidades para as corporações e seus usuários. A interligação destas à Internet permitiu que fossem propostos modelos comerciais baseados nesta tecnologia, com o objetivo de trazer mais valor agregado para as corporações e seus clientes. Este cenário nos parece bem confortável e simples, mas, como qualquer atividade, possui os seus riscos, riscos estes que podem atingir os clientes, as corporações ou ambos.

É neste cenário que a Segurança da Informação tem papel importante, pois o seu objetivo é permitir que se utilizem ferramentas para manter este ambiente seguro e operacional para as corporações e seus clientes.

Não podemos falar em Segurança da Informação sem nos referirmos à Criptografia, que apresenta um conjunto de características necessárias para garantir que as transações e comunicações eletrônicas ocorram de forma segura.

A criptografia é composta por dois tipos de algoritmos: algoritmos simétricos e assimétricos.

1.1 Objetivos

Iremos abordar neste estudo os algoritmos simétricos que apresentam em seu projeto a função de substituição S, conhecida como Caixa-S, estrutura considerada

o coração dos cifradores simétricos e responsável pela segurança do algoritmo.

Neste trabalho é realizado um estudo dos algoritmos e técnicas que foram adotadas para projetar e avaliar as Caixas-S dos principais cifradores de bloco simétricos e efetuar uma comparação entre estes projetos, bem como um estudo dos Algoritmos Genéticos e suas características, sendo este uma das principais linhas de pesquida da área de Computação Evolutiva.

O foco principal deste trabalho, é utilizar a técnica de Algoritmos Genéticos para projetar Caixas-S compatíveis como a do AES e avalia-as através do critério de não linearidade, bem como comparar os resultados alcançados com os conhecidos na literatura.

1.2 Justificativa

A pesquisa de projeto e avaliação de Caixas-S tem sido um dos principais assuntos de estudo dos algoritmos simétricos de criptografia. Neste contexto muitos trabalhos tem sido apresentados com o objetivo de propor técnicas que permitam analisar e projetar Caixas-S, e também verificar se os critérios utilizados em seu projeto atendem as necessidades de segurança necessária desta estrutura. As técnicas de projetos de Caixas-S tem sofrido uma grande evolução, passando por randômica, randômica com testes, uso de matemática elementar e baseada em princípios matemáticos[1]. Estas técnicas buscam projetar Caixa-S seguras, ou seja, que apresentem boa difusão[2]. Os estudos apresentados nos últimos tempos tem se concentrado em técnicas baseadas em princípios matemáticos, ou seja, o estudo de funções que apresentem alto grau de difusão.

[1] É classificado como matemática elementar a utilização dos operadores $+,-,*,OR,XOR$ e princípios matemáticos a utilização de funções.

[2] Princípio proposto por Shannon no contexto da criptografia.

1.3 Motivação

Estudar as técnicas de projeto de Caixas-S tem sido muito importantes para o melhor entendimento dos cifradores estudados, poder verificar e analisar os critérios utilizados nestes grandes projetos tem tornado este estudo muito interessante. Poder aplicar o uso de Algoritmos Genéticos no projetos de Caixas-S tem se apresentado com um grande desafio, desafio muito interessante e de grande valia para o estudo destas técnicas de projeto.

1.4 Materiais e Métodos

A monografia foi escrita em Latex utilizando o editor de texto e compilador WinEdt, ferramenta desenvolvida para o ambiente gráfico Windows e estilo desenvolvido e mantido pelo GruTeX (http://www.inf.ufsc.br/grutex), para as dissertações de mestrado e teses de doutorado do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Para desenvolvimento da implementação foi utilizada a linguagem C++ e o paradigma de orientação a objetos, o compilador utilizado foi o Microsoft Visual C++ em função do ambiente disponível para testes.

Para realização dos teste foi utilizado um laboratório com máquinas Pentium III 400 Mhz com 128 Mb de memória RAM e sistema operacional Windows 2000 Server, quando este se encontrava disponível.

1.5 Trabalhos Correlacionados

A utilização de AGs na área de Criptografia se iniciou com trabalhos de criptoanálise dos cifradores clássicos [MAT 93, SPI 93], os AGs [MAT 93, SPI 93].

Nos trabalhos dos pesquisadores do Information Security Research Center da Universidade de Queensland, Austrália, os AGs tem sido utilizados para projetar Caixas-S com alto grau de difusão [WM 97, WM 98, WM 99].

No trabalho de Burnett [LB 00], são utilizados as técnicas de Hill Climbing (subida de encosta [LAC 99]) e AGs para projetar Caixas-S compatíveis com o cifrador MARS, e eles provaram que as Caixas-S geradas são superiores as Caixas-S geradas pelos projetistas do MARS.

1.6 Conteúdo da Dissertação

Este trabalho encontra-se organizado da seguinte forma: O capítulo 2 descreve os fundamentos da criptografia, as técnicas utilizadas, a estrutura de Feistel, apresenta os principais cifradores modernos e os candidatos ao AES que possuem em sua estrutura Caixas-S e, finalmente, faz-se uma comparação entre estes cifradores. O capítulo 3 apresenta o histórico das Caixas-S, suas principais características e um resumo contendo as técnicas utilizadas para projetar os principais cifradores modernos e os candidatos ao AES. O capítulo 4, apresenta uma introdução ao conceito de não linearidade, funções booleanas, ferramentas matemáticas utilizadas para efetuar o cálculo da não linearidade e os principais resultados conhecidos. O capítulo 5 apresenta uma introdução aos Algoritmos Genéticos, a terminologia adotada por esta técnica, sua simulação do processo de evolução, elementos e operadores utilizados, suas principais aplicações e a aplicação em Criptografia. No capítulo 6, é proposta a utilização de Algoritmos Genéticos para projetar Caixas-S para os cifradores de bloco simétricos, são explicadas as estratégias que se pretende adotar para alcançar este objetivo, e apresentada a implementação realizada e os resultados obtidos. No capítulo 7, apresentamos as considerações finais sobre o trabalho e informações sobre trabalhos futuros. No anexo A, temos exemplos de Caixas-S geradas usando a técnica proposta. No anexo B, temos os fontes desenvolvidos em linguagem C++ da implementação realizada neste projeto de pesquisa.

Capítulo 2

Cifradores de Bloco Simétricos

2.1 Introdução

O objetivo deste capítulo é apresentar os conceitos referentes à Criptografia, mais especificamente, realizar um estudo dos principais Cifradores de Bloco Simétricos que apresentam em sua estrutura Caixas-S.

Começamos fazendo uma breve introdução à Criptografia, apresentando as técnicas Clássicas e Modernas. Em seguida, fazemos um estudo da estrutura de Feistel que é a base dos principais cifradores de bloco simétricos. Apresentamos também um estudo do funcionamento dos principais algoritmos que possuem Caixas-S em sua estrutura e finalizamos com uma comparação contendo suas principais características.

2.2 Introdução à Criptografia

Segundo [NAI 99], A Criptografia é a ciência que lança mão de recursos matemáticos para cifrar e descifrar dados, sendo a Criptoanálise a ciência que efetua a análise e busca quebrar (encontrar vulnerabilidades) nos algoritmos de criptografia, sendo ambas englobadas pela Criptologia.

Os algoritmos criptográficos utilizados para realizar estas atividades são conhecidos como *cifras*. Geralmente possuem duas funções relacionadas: uma para cifrar

e outra para descifrar [SCH 96]. A função de cifrar consiste em transformar um texto aberto em texto cifrado, a função de descifrar é a sua operação inversa, ou seja, consiste em transformar um texto cifrado em texto aberto. Conforme é ilustrado na figura 2.1.

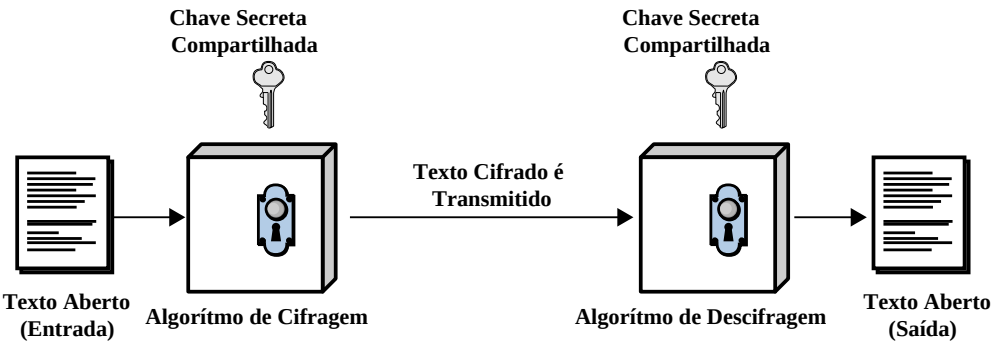

Figura 2.1: Processo de Cifragem e Descifragem: O texto aberto é fornecido como entrada para o algoritmo de cifragem, que utiliza-se de uma chave secreta compartilhada entre as partes interessadas, como resultado temos o texto cifrado. No receptor é utilizado o algoritmo de descifragem em conjunto com a chave secreta, como resultado temos o texto em sua forma original (aberto) [STA 99].

A criptografia é a ferramenta utilizada para prover [MEN 97]: Confidencialidade; Integridade dos Dados; Autenticação; Não repúdio.

2.3 Técnicas Clássicas

As mais variadas técnicas de criptografia vêm sendo usadas desde os tempos mais remotos (técnicas clássicas) até a atualidade (técnicas modernas).

As técnicas clássicas são baseadas em substituições e transposições e as técnicas modernas em algoritmos de criptografia [SCH 96, STA 99].

2.3.1 Substituição

A técnica de substituição consiste na simples troca de letras do texto aberto por outras letras, números ou símbolos.

Estes cifradores são classificados em quatro tipos [SCH 96]:

Substituição Simples ou Monoalfabética: É aquela em que cada letra do texto aberto corresponde a outra no texto cifrado. O famoso cifrador de César é baseado nesta técnica.

Substituição Homofônica: É o tipo de substituição em que uma letra do texto aberto pode corresponder a mais de um no texto cifrado. Ex.: *A* corresponde a 1, 9 e 11. O cifrador usado pelo Duque de Mantua utiliza-se desta técnica.

Substituição em Grupos: É o tipo de substituição em que blocos de letras do texto aberto são criptografados juntos. Ex.: *ASA* pode corresponder a *BDP* ou *SSA* pode corresponder a *LLF*. O cifrador Hill, desenvolvido por Lester Hill, em 1929, se utiliza desta técnica.

Substituição Múltipla ou Polialfabética: É aquela em que a substituição simples é aplicada sobre a mesma mensagem mais de uma vez. Esta técnica era utilizada pala *Union Army* durante a guerra civil Americana. O cifrador Vigenère é um exemplo desta técnica.

2.3.2 Transposição

A técnica de transposição consiste na simples permutação dos caracteres, ou seja, o texto aberto continua o mesmo, mas a ordem dos seus caracteres é alterada.

Esta técnica preserva a incidência do mesmo número de caracteres e a sua freqüência o que o torna fácil de ser criptoanálisado [MEN 97, STA 99].

Este tipo de algoritmos foi utilizado pelo cifrador Alemão ADFGVX durante a I Guerra Mundial [SCH 96].

2.4 Técnicas Modernas

O que chamamos de técnicas modernas são os algoritmos de criptografia que estão sendo utilizados atualmente. Estes algoritmos são conhecidos como algoritmos convencionais.

Os algoritmos convencionais são geralmente públicos e as partes que desejam se comunicar usam uma chave secreta que deverá ser utilizada em conjunto com o algoritmo. A chave secreta consiste de uma *string* randômica de bits de tamanho variável (40 bits, 56 bits, 64 bits, 128 bits, 192 bits e 256 bits). Atualmente, é aconselhável a adoção de chaves de 128 bits [SCH 00].

Os algoritmos modernos são classificados em:

Simétricos: São os algoritmos que se utilizam de uma chave compartilhada pelo emissor (Alice) e receptor (Bob) usada para cifrar e descifrar, ou seja, a chave usada para cifrar é a mesma usada para descifrar e Alice e Bob devem conhecer a chave secreta.

Assimétricos: São os algoritmos que se utilizam de duas chaves, ou seja, uma chave é utilizada para cifrar e a outra para descifrar. Estes algoritmos são conhecidos como algoritmos de chave pública.

A segurança dos algoritmos simétricos esta baseada nos seguintes fatores: a força do algoritmo e tamanho de chave usada [SCH 96].

2.4.1 Tipos de Algoritmos Simétricos

Existem basicamente duas classes de algoritmos simétricos que são largamente discutidas: Bloco e Fluxo [SCH 96, MEN 97, STA 99].

2.4.2 Cifrador de Bloco Simétrico

Um cifrador de bloco é aquele que, dada a mensagem (texto aberto), divide a mensagem em blocos de tamanho fixo e efetua a cifragem gerando uma mensagem (texto cifrado) de mesmo tamanho.

Devido a esta característica, o tamanho do bloco usado pelo cifrador passa a ser mais uma das importantes características na segurança deste tipo de algoritmo [ROB 95].

A grande maioria dos algoritmos de criptografia simétricos utiliza esta técnica [MEN 97].

2.4.3 Cifrador de Fluxo Simétrico

Os cifradores de fluxo simétricos são um caso particular muito importante do cifrador de bloco, pois o tamanho do seu bloco é de 1 bit. Neste caso, cada bit da mensagem (texto aberto) irá sofrer a cifragem gerando um bit de mensagem (texto cifrado).

Esta técnica é muito importante para os casos em que:

- Os dados devem ser processados bit a bit;
- O equipamento não suporta buferização;
- O equipamento possui pouca memória.

Embora os cifradores de bloco e cifradores de fluxo sejam diferentes, cifradores de bloco podem ser implementados como cifradores de fluxo e vice versa [SCH 96].

2.5 Projeto de Cifradores de Bloco

Teoricamente, todos os algoritmos de criptografia de bloco simétricos em corrente uso são baseados na estrutura de um cifrador de bloco conhecido como Rede de Feistel.

Um cifrador de blocos opera com blocos de texto aberto de *n* bits para produzir blocos cifrados de *n* bits. Neste processo, existem 2^n possíveis diferentes blocos de texto aberto. Para que o processo de cifrar seja reversível (ser possível descifrar), cada

bloco precisa produzir um único bloco cifrado. Esta transformação é conhecida como reversível ou não singular.

Este tipo de sistema, quando usado com um bloco de tamanho pequeno, é equivalente ao sistema clássico de substituição, ou seja, é fácil de ser criptoanálisado. Se utilizarmos blocos de tamanho grande, iremos enfrentar problemas de implementação e desempenho dos algoritmos. Estes aspectos foram observados por Feistel [FEI 73], que apresentou uma alternativa para o problema do trabalho com blocos de tamanho grande.

2.5.1 Cifrador de Feistel

Feistel propôs que podemos nos aproximar de um cifrador de substituição simples utilizando o conceito de cifrador de produto, que consiste em combinar dois ou mais cifradores básicos em seqüência, de forma que o resultado final ou produto é criptograficamente mais forte que qualquer um dos cifradores envolvidos [ROB 95, MEN 97, STA 99].

Feistel sugeriu a utilização de cifradores que alternam substituições e permutações, estas características já haviam sido propostas por Claude Shannon para o desenvolvimento de cifradores de produto que tivessem boa resistência à criptoanálise estatística. Estas características são conhecidas como confusão e difusão.

Estas características tornaram-se os pilares dos cifradores de bloco modernos [ROB 95, STA 99].

Confusão: É a técnica que consiste em tornar a relação entre as estatísticas do texto aberto, texto cifrado e o valor da chave a mais complexa possível.

Difusão: É a técnica que consiste em dissipar a estrutura estatística do texto aberto, ou seja, visa tornar as relações entre o texto aberto e o texto cifrado as mais complexas possíveis.

Atualmente, este tipo de estrutura é conhecida como SPN (Rede de Substituição-Permutação[1]) [SCH 96, MEN 97, STA 99].

[1] Substitution-Permutation Network

2.5.2 Estrutura do Cifrador de Feistel

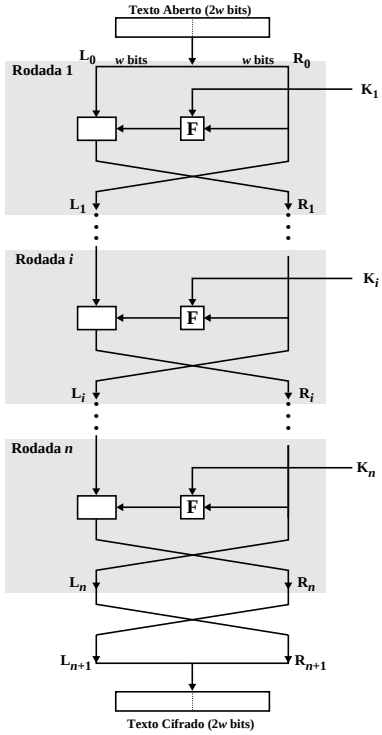

Figura 2.2: Estrutura Clássica de Feistel: É fornecido um texto plano de $2w$ bits como parâmetro de entrada da primeira rodada, este é dividido em duas partes iguais, a parte da esquerda e direita, a parte da direita passa pela função F em conjunto com chave da rodada, após este processo as duas partes são trocadas, a parte da direita passa para a esquerda e a esquerda passa para a direita, após a troca é iniciado uma nova rodada com estas partes como parâmetros. Após todas as rodadas é feita uma nova troca de partes e temos o texto cifrado de $2w$ bits [STA 99].

A figura 2.2 representa a estrutura clássica de Feistel com sua SPN.

O processo de cifrar desta estrutura clássica é ilustrada pela figura 2.3(a) e é realizada da seguinte forma:

- As entradas para o algoritmo são o texto aberto com tamanho de bloco de 2*w* bits e chave *K*.
- O texto aberto é dividido em duas partes de *w* bits, LE_0 e RE_0.

- A parte LE_0 sofre um $\bigoplus$ (OU exclusivo) com a parte RE_0 que passou pela função F. O resultado desta operação [$LE_0 \bigoplus$ F(RE_0,K_1)] servirá de entrada para a próxima rodada (LE_{i-1}). Nesta passagem é efetuada a substituição.

- A parte RE_0 será mantida intacta e servirá de entrada para a próxima rodada (RE_{i-1}), somente sendo trocado de lado. Esta troca entre o lado esquerdo e direito caracteriza a permutação.

- Este processo se repete várias vezes para produzir o bloco de texto cifrado. Cada parte do processo é conhecida como rodada.

- A cada rodada, é calculada uma nova subchave (K_{i-1}) a partir da chave anterior (K_i).

O processo de descifrar desta estrutura clássica é ilustrada pela figura 2.3(b) e é realizada da seguinte forma:

- As entradas para o algoritmo são o texto cifrado com tamanho de bloco de 2*w* bits e as subchaves K_n em ordem inversa, ou seja, a primeira chave a ser usada será K_n=K_{16}, a segunda chave a ser usada será K_{n-1}=K_{15}, assim sucessivamente até chegarmos a K_1.

- O texto cifrado é dividido em duas partes de *w* bits, LD_0 e RD_0.

- A parte LD_0 sofre um $\bigoplus$ (OU exclusivo) com a parte RD_0 que passou pela função F. O resultado desta operação [$LD_0 \bigoplus$ F(RD_0,K_{16})] servirá de entrada para a próxima rodada (LD_{i-1}) .

- A parte RD_0 será mantida intacta e servirá de entrada para a próxima rodada (RD_{i-1}), somente sendo trocado de lado.

- Este processo se repete várias vezes para produzir o bloco de texto aberto. Cada parte do processo é conhecida como rodada.

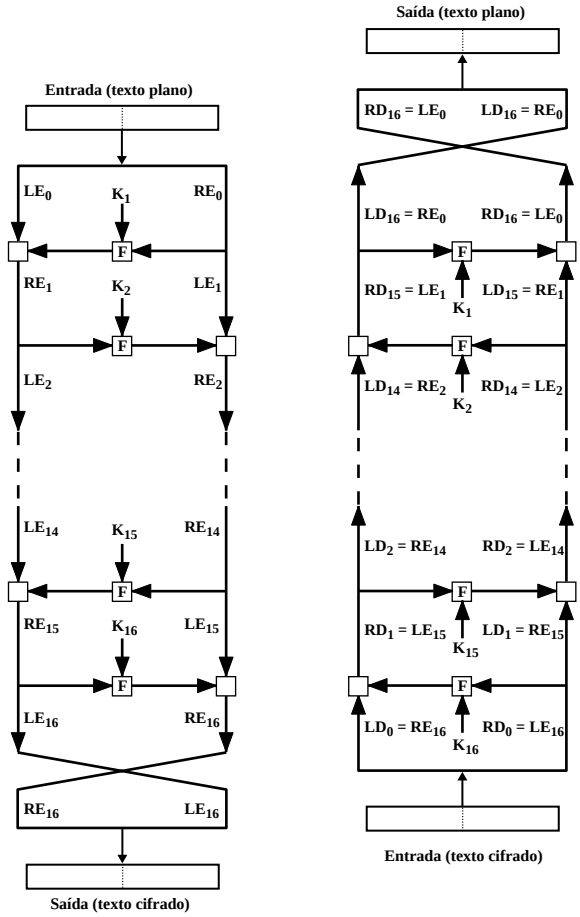

Figura 2.3: Estrutura de Feistel - a) Cifrar e b) Descifrar: No processo de cifrar (a) é fornecido o texto plano para ser cifrado ($2w$ bits), o mesmo atravessa todas as rodadas e por final fornece como saída o texto cifrado. No processo de descifrar (b) o processo é realizado com as suas chaves em ordem inversa e nos fornece como saída o texto aberto [STA 99].

Podemos notar que o processo de descifrar é basicamente o mesmo que o de cifrar, esta característica é muito importante, pois nos permite utilizar o mesmo algoritmo para realizar ambas as tarefas [ROB 95].

A exata implementação de uma estrutura de Feistel depende da escolha dos seguintes parâmetros e características de projeto [STA 99]:

Tamanho do Bloco: Grandes blocos são mais seguros, mas afetam a performance do

algoritmo. O tamanho de bloco utilizado largamente pelos cifradores de bloco é de 64 bits.

Tamanho da Chave: Grandes chaves são mais seguras, mas afetam a performance do algoritmo. As chaves menores ou iguais a 64 bits são atualmente consideradas inseguras [SCH 00].

Número de Rodadas: O cifrador de Feistel, quando usado com uma simples rodada, é inseguro, mas múltiplas rodadas aumentam a sua segurança. A quantidade típica de rodadas é 16.

Geração de Subchaves: Quanto mais complexo for este algoritmo, maior será a dificuldade de criptoanálise.

Função da Rodada: Quanto mais complexo for este algoritmo, maior será a dificuldade de criptoanálise.

2.6 Principais Cifradores de Bloco Simétricos

Nesta seção, iremos apresentar as características dos principais cifradores de bloco simétricos. Estes cifradores foram escolhidos devido ao fato de serem projetos bastante estudados, pesquisados pela comunidade científica e possuírem em seu projeto Caixas-S.

2.6.1 DES - Data Encryption Standard

O DES, conhecido como Data Encryption Algorithm (DEA) pela ANSI e DEA-1 pela ISO, foi o padrão de criptografia durante 20 anos. Embora já seja considerado inseguro em muitas aplicações, devido a sua chave de 56 bits, resistiu bem todos estes anos à criptoanálise e manteve-se seguro [SCH 96].

2.6.1.1 Histórico

Na década de 60, a IBM iniciou o projeto de pesquisa na área de Criptografia, liderado por Horst Feistel.

O resultado desta empreitada foi o desenvolvimento do algoritmo chamado LUCIFER, em 1971, que foi utilizado em um projeto para o Banco de Londres pela IBM. LUCIFER era um cifrador de bloco baseado na estrutura de Feistel que opera com blocos de 64 bits e chaves com tamanho de 128 bits, utiliza de operações lógicas simples em pequenos grupos de bits e que pode ser implementado com muita eficiência em hardware.

Em 1972, o NBS, atual NIST, iniciou o programa para proteção de computadores e comunicação de dados. Como parte deste programa, constava a adoção de um algoritmo simples e padrão de criptografia.

Em 1973, o NBS publicou uma solicitação de proposta para a escolha do cifrador padrão nacional (EUA). Nesta primeira solicitação nenhum dos candidatos atendia os pré-requisitos, em 1974 foi publicada uma nova solicitação, quando receberam a proposta de um bom candidato, o algoritmo LUCIFER.

O NBS solicitou a ajuda do NSA para avaliar a segurança do algoritmo e determinar se este atendia os pré-requisitos para tornar-se o padrão nacional.

Em 1975, foram publicados os detalhes do algoritmo e solicitados à comunidade comentários sobre o mesmo.

A redução do tamanho da chave de 128 bits para 56 bits, por parte do NSA, gerou muitas críticas e suspeitas sobre o motivo da modificação, que na época não foi explicada.

As críticas se concentravam e ainda se concentram sobre dois fatores do algoritmo:

Tamanho da Chave: A sua redução poderia tornar o algoritmo suscetível ao ataque da força bruta, que consiste em testar exaustivamente todas as possíveis chaves.

Projeto das Estruturas de Substituição (Caixas-S): Devido à implementação destas estruturas suspeita-se sobre a possibilidade da inclusão de vulnerabilidades que pos-

sibilitassem ao NSA descifrar a mensagem. Estas estruturas são responsáveis em prover a não linearidade ao algoritmo.

Apesar de todas as críticas, o algoritmo foi eleito o DES (Data Encryption Standard - Padrão para Criptografia de Dados), em 1976, pelo NBS, por cinco anos. Desta adoção surgiram os documentos FIPS PUB 46 [FIP 77], em 1977 e FIPS PUB 81 [FIP 80], em 1980. Tornando-se o primeiro algoritmo comercial moderno a ter toda sua especificação de implementação detalhada, um padrão aberto.

Nas avaliações de 1983, 1988 e 1993, o DES manteve-se como padrão. Em 1997, o NIST abriu uma chamada para propostas ao novo padrão como os seus respectivos requisitos, este processo chamou-se AES (Advanced Encryption Standard). Em 1998, teve início o AES e o DES novamente manteve-se como padrão, mas já com os dias contados, valendo até o dia em que o processo AES terminasse, o que aconteceu em 02 de Outubro de 2000, com a aprovação do algoritmo Rijndael como novo padrão de criptografia pelo NIST [NIS 00].

2.6.1.2 Descrição do Algoritmo

O DES é um cifrador de bloco interativo do tipo Feistel de 16 rodadas, que cifra blocos de 64 bits com uma chave de 56 bits [ROB 95, SCH 96, MEN 97, STA 99].

Na figura 2.4, é ilustrado o diagrama do cifrador de blocos - DES.

Cada rodada do DES implementa as características propostas por Shannon (confusão e difusão), através de operações simples de substituição seguidas de permutação, este processo se repete por 16 vezes no algoritmo.

Basicamente, o algoritmo é composto das seguintes etapas, conforme é ilustrado na figura 2.5:

Entrada do Texto Aberto e Chave: O bloco de 64 bits de texto aberto a ser cifrado é informado, bem como a chave de 56 bits, que passa por uma permutação através de uma tabela conhecida como PC-1 (Permutação 1).

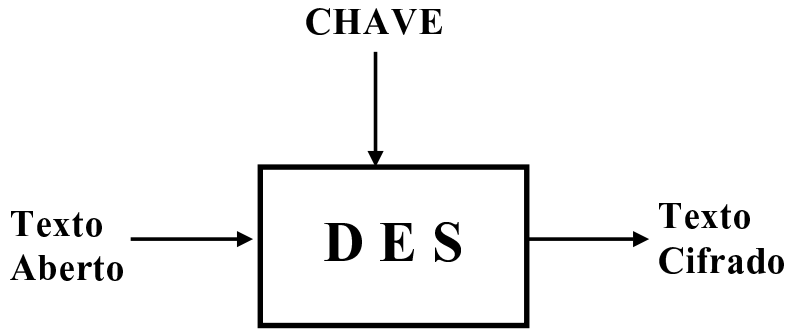

Figura 2.4: Cifrador de Bloco - DES: O algoritmo recebe como parâmetros o texto plano e a chave e nos fornece como saída o texto cifrado.

Permutação Inicial: É utilizada para rearranjar os bits de entrada, gerando uma nova seqüência de entrada, ou seja, este rearranjo é feito com a ajuda da tabela conhecida como Permutação Inicial (IP), assim obtém-se a nova seqüência. Segundo a Tabela 2.1, o bit 58 de entrada passará a ser o bit 1 na nova entrada, o bit 50 de entrada passará a ser o bit 2 na nova entrada, assim sucessivamente para os demais bits de entrada. Segundo Schneier [SCH 96], a permutação inicial e final não afetam a segurança do algoritmo.

Tabela 2.1: Permutação Inicial - IP

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	16	7

Após a Permutação Inicial, o texto é divido em duas partes de 32 bits, esquerda(L_0) e direita(R_0).

Rodada 1 até 16: Esta roda também é conhecida como função F. Em cada rodada é realizada a combinação da parte da direita(R_0) com a subchave gerada para esta, através de expansões, substituições e permutações.

Troca de 32-bit: Os 64 bits gerados após todas as rodadas sofrem uma troca, a parte da

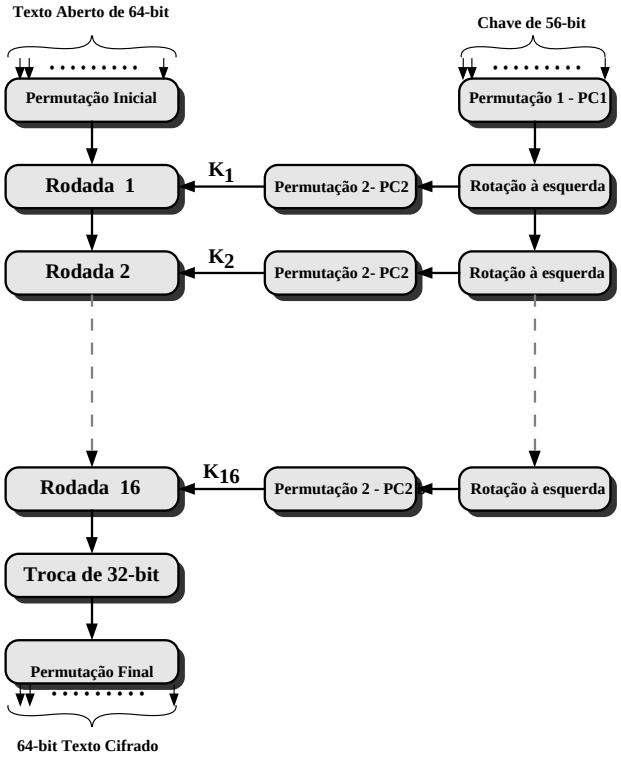

Figura 2.5: Diagrama Completo do Algoritmo DES: O DES recebe como entrada o texto plano de 64 bits que sofre permutação, em seguida passa pelas 16 rodadas, que recebem como entrada os 64 bits e a chave, que foi gerada recebendo 56 bits como entrada que sofreu permutação, rotação a esquerda e nova permutação. No final os 64 bits sofrem substituição e permutação e fornecem como saída o texto cifrado [STA 99].

esquerda(L_0) passa para a parte da direita(R_0) e vice-versa.

Permutação Final: Assim como a permutação inicial, esta irá passar os 64 bits gerados por uma tabela (Tabela 2.2) conhecida como Permutação Final (IP^{-1}), que é a inversa da permutação inicial, gerando 64 bits em uma nova disposição.

Saída do Texto Cifrado: Temos como produto final o texto cifrado de 64 bits.

Iremos fazer uma análise mais detalhada de uma Rodada do algoritmo, conforme ilustração da figura 2.6.

Funcionalmente todas as rodadas são equivalentes. O procedimento

Tabela 2.2: Permutação Final - IP^{-1}

40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

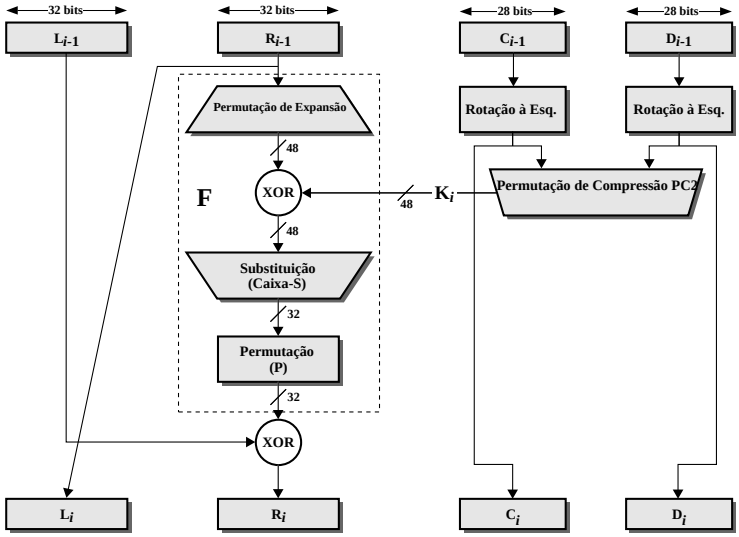

Figura 2.6: Rodada Detalhada do Algoritmo DES: A parte da esquerda de 32 bits passa para a direita na próxima rodada após sofrer XOR com o resultado da parte da direita que passou pelas fases de Permutação de Expansão, XOR, Caixas-S, Permutação. A parte da direita passa a ser a parte da esquerda na próxima rodada. Para cada rodada as chaves sofrem permutação à esquerda e direita e finalmente permutação com compressão [STA 99].

consiste em pegar 32 bits como entrada L_{i-1} e R_{i-1} vindos da rodada anterior e produzindo L_i e R_i para $1 \leq i \leq 16$. Estas operações podem ser representadas da seguinte forma:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \bigoplus f(R_{i-1}, K_i),$$

onde, $f(R_{i-1}, K_i) = P(S(E(R_{i-1}) \bigoplus K_i))$.

Dentro de cada rodada nós temos as seguintes operações:

Permutação de Expansão: Esta operação faz a expansão da parte da direita (R_i) de 32 bits para 48 bits, com base na tabela 2.3 e tem como objetivo deixar a parte da direita com o mesmo tamanho da subchave e ter mais bits como entrada para as Caixas-S.

Tabela 2.3: Permutação com Expansão

32	1	2	3	4	5	4	5	6	7	8	9
8	9	10	11	12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21	22	23	24	25
24	25	26	27	28	29	28	29	30	31	32	1

Geração de Subchaves: A rodada K_i recebe uma chave de 56 bits após a passagem pela tabela PC-1, que é dividida em duas partes C_{i-1} (esquerda) e D_{i-1} (direita), de 28 bit cada. Estas partes sofrem rotação à esquerda que são calculadas em função da rodada em que a geração se encontra, com base na tabela 2.4. Estes novos valores rotacionados servem de entrada para a próxima rodada e garantem que um novo conjunto de subchaves será gerado. Após a passagem pela tabela 2.4, os bits passam pela tabela 2.5 que tem por objetivo selecionar 48 bits dos 56 bits enviados, a subchave passa a ter o tamanho de 48 bits.

Tabela 2.4: Rotação à Esquerda

Número da Rodada	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bits a rotacionar	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Tabela 2.5: Permutação com Compressão

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

OU Exclusivo: É realizado um XOR com o resultado da Permutação de Expansão e a subchave gerada, resultando em um bloco de 48 bits.

Substituição (Caixas-S): Nesta etapa é realizada uma substituição através de 8 Caixas-S (Tabela 2.6) ou, simplesmente, Caixas-S. Os 48 bit recebidos são divididos em 8 sub-blocos de 6 bit. Cada sub-bloco irá passar por uma Caixa-S separada, o primeiro sub-bloco passará na Caixa-S_1, o segundo sub-bloco passará na Caixa-S_2 e, assim, sucessivamente. Durante essa passagem pela Caixa-S, cada sub-bloco de 6 bit de entrada gera um sub-bloco de 4 bit, produzindo no final 32 bit (8 sub-blocos de 4 bit), conforme é ilustrado na figura 2.7.

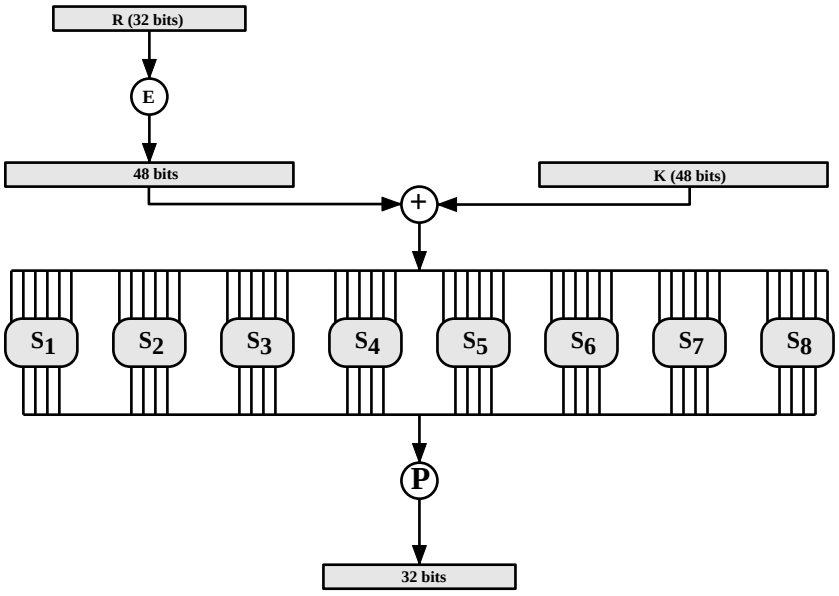

Figura 2.7: Caixas-S: É fornecido como parâmetro de entrada 48 bits que são divididos em 8 grupos de 6 bits, cada grupo passa por uma Caixa-S e fornece como saída 4 bits que compõe os 32 bits gerados.

Permutação (Caixa-P): Os 32 bit resultantes das Caixas-S sofrem uma permutação de acordo com a tabela 2.7. Esta permutação é conhecida como permutação direta [STA 99].

OU Exclusivo: A parte resultante da Caixa-P sofre um XOR com a metade da esquerda L_{i-1} que até este momento não havia sido utilizada.

Troca: As partes da esquerda L_{i-1} e direita R_{i-1} são trocadas de lugar e se inicia uma

Tabela 2.6: Caixas-S de Substituição

	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
Caixa-S_1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
Caixa-S_2	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
Caixa-S_3	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
Caixa-S_4	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
Caixa-S_5	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	1
Caixa-S_6	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	3
	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
Caixa-S_7	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
Caixa-S_8	1	15	3	8	10	3	7	4	12	5	6	11	0	14	9	2
	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

nova rodada.

O processo de descifrar neste algoritmo é o mesmo, somente sendo necessária a utilização das chaves em ordem inversa (K_{16},K_{15}, ...,K_1).

Tabela 2.7: Permutação - Caixa-P

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

2.6.2 CAST

O CAST foi desenvolvido no Canadá por Carlisle Adams e Stafford Tavares. O nome, acredita-se a principio, é composto pelas iniciais dos autores, apesar de não admitirem isso [SCH 96].

O CAST é um algoritmo simétrico baseado na estrutura clássica de Feistel, com sua rede de substituições e permutações, e opera com blocos de 128 bits em quatro rodadas, produzindo um texto cifrado de 128 bits. Permite a utilização de chaves de 128, 192, 224 e 256 bits. Utiliza-se de quatro Caixas-S. Apresenta boa resistência à criptoanálise diferencial e linear e possui outras características desejadas por sistemas criptográficos, como: efeito avalanche, SAC, BIC, não é complementar, sem chaves fracas ou semi-fracas [ADA 99].

Este algoritmo apresenta duas grandes diferenças com relação à estrutura de Feistel:

Subchaves: Utiliza-se de duas subchaves em cada rodada.

Função F: É aplicada de forma diferente em função de sua rodada.

Os três tipos de função F utilizados pelo algoritmo CAST encontram-se definidos na tabela 2.8.

Tabela 2.8: Função F - Algoritmo CAST

Tipo 1 (f1)	$I = ((Km_i + D) <<< Kr_i)$ $F = ((S1[Ia] \bigoplus S2[Ib]) - S3[Ic]) + S4[Id]$
Tipo 2 (f2)	$I = ((Km_i \bigoplus D) <<< Kr_i)$ $F = ((S1[Ia] - S2[Ib]) + S3[Ic]) \bigoplus S4[Id]$
Tipo 3 (f3)	$I = ((Km_i - D) <<< Kr_i)$ $F = ((S1[Ia] + S2[Ib]) \bigoplus S3[Ic]) - S4[Id]$

Onde:

D = Dado de entrada para a operação.

Km_i = Subchave da rodada.

Kr_i = Subchave da rodada.

$S1$,$S2$,$S3$,$S4$ = Caixa-S que é utilizada.

Ia = Primeiro bloco de 4 bit dos 32 bit intermediários.

Ib = Segundo bloco de 4 bit dos 32 bit intermediários.

Ic = Terceiro bloco de 4 bit dos 32 bit intermediários.

Id = Quarto bloco de 4 bit dos 32 bit intermediários.

$<<<$ = Rotação à esquerda.

As características apresentadas acima fazem parte do CAST-128 que é a base para o projeto do CAST-256, estas características foram mantidas na íntegra no projeto do CAST-256 [ADA 99].

O processo de cifragem do CAST-256 ocorre da seguinte forma:

BETA $=$ 128 bits de texto aberto.

for (i=0; i $<$ 6; i++)

BETA $<-$ Q_i(BETA)

for (i=0; i $<$ 12; i++)

BETA $<-$ $QBAR_i$(BETA)

128 bits de texto cifrado $=$ BETA.

Onde:

BETA $=$ (ABCD), sendo A, B, C e D blocos de 32 bits.

A operação BETA $<-$ Q_i(BETA) é a seguinte:

$C = C \bigoplus f1(D,Kr0_i,Km0_i)$

$B = B \bigoplus f2(C,Kr1_i,Km1_i)$

$A = A \bigoplus f3(B,Kr2_i,Km2_i)$

$D = D \bigoplus f1(A,Kr3_i,Km3_i)$

A operação BETA $<-$ $QBAR_i$(BETA) é a seguinte:

$D = D \bigoplus f1(A,Kr3_i,Km3_i)$

$A = A \bigoplus f3(B,Kr2_i,Km2_i)$

$$B = B \bigoplus f2(C,Kr1_i,Km1_i)$$

$$C = C \bigoplus f1(D,Kr0_i,Km0_i)$$

O processo de descifrar é exatamente o mesmo, com exceção das chaves que devem ser utilizadas em ordem reversa.

Na figura 2.8 é apresentada, com detalhes, uma rodada do algoritmo.

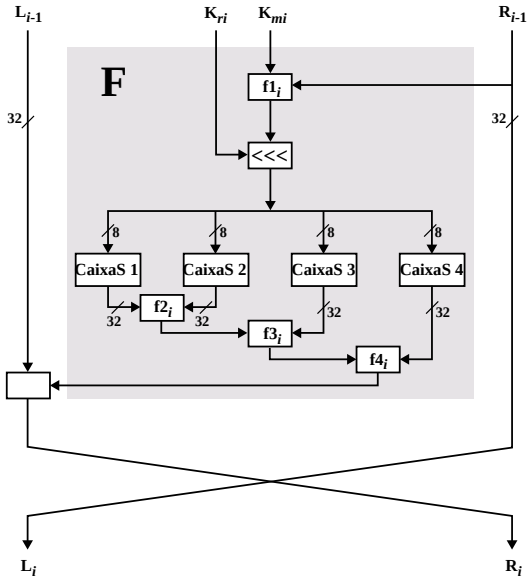

Figura 2.8: Função F - Algoritmo CAST: A parte da direita passa a ser a parte da esquerda para a próxima rodada e passa pela função f_1, rotação à esquerda, é divido e passa por 4 Caixas-S que passam pelas funções f_2, f_3 e f_4, após este processo sofre nova operação com a parte da esquerda e passa a ser a parte da direita da próxima rodada [STA 99].

O CAST possui um processo de geração de chaves muito complexo e bem diferente dos utilizados em outros cifradores simétricos, ele foi concebido para que estas chaves sejam resistentes à criptoanálise.

O CAST se utiliza de Caixas-S fixas, sendo o projeto das Caixas-S uma de suas boas características [SCH 96, STA 99]. São concebidas utilizando-se funções *Bent*[2] parciais e permitem a geração de Caixas-S com boa não linearidade [ROB 95].

[2]São funções Booleanas $g(x) : Z_2^n \rightarrow Z_2,\ n = 2l$, na qual todos os coeficientes da transformada de

O CAST também participou como candidato (CAST-256) ao novo AES [NIS 00], sendo eliminado na segunda rodada.

2.6.3 MARS

O MARS foi o projeto apresentado pela IBM como candidato ao novo AES, ficando entre os cinco algoritmos finalistas [NIS 00], sendo desenvolvido por Carolynn Burwick, Don Coppersmith, Edward D/'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas, Luke O/'Connor, Mohammad Peyravian, David Safford e Nevenko Zunic.

O MARS é um algoritmo simétrico baseado na estrutura clássica de Feistel[3] do tipo 3 opera com blocos de 128 bit, produzindo um texto cifrado de 128 bits. Permite a utilização de chaves de (128 a 448 bit), variando em módulos de 32 bits.

O algoritmo foi projetado desta forma para poder resistir muito bem a todos os ataques conhecidos baseados na técnica de criptoanálise, e suas rodadas provêm bom efeito avalanche, oferecendo uma boa margem de segurança contra os ataques conhecidos e que podem surgir [BUR 99].

O algoritmo utiliza-se de apenas uma Caixa-S com 512 entradas, de 32 bits, sendo esta Caixa-S também utilizada pelo processo de expansão de chaves, em alguns casos a Caixa-S utilizada é dividida em duas partes S_0 e S_1 com 256 entradas de 32 bits cada (8×32), esta Caixa-S escolhida apresenta uma boa resistência aos ataques da criptoanálise diferencial e linear e também apresenta um bom efeito avalanche para os dados e chaves [BUR 99, COP 99]. Utiliza-se de uma Caixa-S fixa [COP 99].

O algoritmo MARS foi projetado para operar em três fases, como podemos observar na figura 2.9:

Em cada fase da figura 2.9, temos as seguintes operações:

Fourier $G(w)$ de $(-1)^{g}(x)$ são definidas para todo $w \ \epsilon \ Z_2^n$ com $G(w) = \frac{1}{\sqrt{2^n}} \sum_{x \ \epsilon \ Z_2^n} (-1)^{g(x) + x \bullet w}$ e magnitude unitária $\left|G(w) = 1\right|$. A operação $x \bullet w$ é o produto de x e w sobre Z_2^n

[3] É a estrutura na qual em cada rodada uma palavra fornecida e algumas palavras da chave, são usadas para modificar todas as outras palavras fornecidas. No tipo 1 em cada rodada uma palavra fornecida é usada para modificar somente uma outra palavra fornecida.

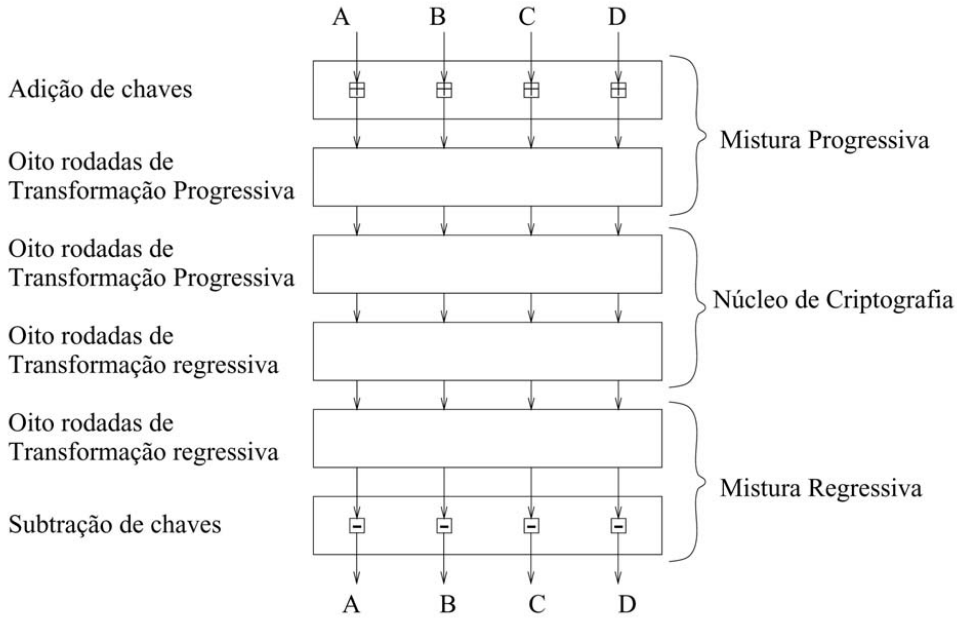

Figura 2.9: Estrutura do Cifrador MARS: O algoritmo recebe 4 palavras de 32 bits como entrada, estas palavras alimentam as 3 etapas do algoritmo, Mistura Progressiva, Núcleo de Criptografia e Mistura Regressiva. Após estas etapas o texto cifrado é fornecido (4 palavras de 32 bits) [BUR 99].

Mistura Progressiva: Nesta fase, primeiramente, a chave é adicionada a cada dado recebido como entrada, em seguida, são executadas oito rodadas da rede de Feistel tipo 3. Em cada rodada, o dado recebido (uma palavra de 32-bits, conhecida como origem) é utilizado para modificar os outros três dados recebidos (conhecidos como palavras destino). Os quatro bytes da palavra origem b_0,b_1,b_2,b_3 são utilizados como índices para as Caixas-S S_0 e S_1, da seguinte forma: b_0,b_2 são os índices para a Caixa-S S_0 e b_1,b_3 são os índices para a Caixa-S S_1. Primeiramente, ocorrem as seguintes operações: XOR de $S_0[b_0]$ com a primeira palavra de destino recebida, adição de $S_1[b_1]$ com a primeira palavra de destino recebida, adição de $S_0[b_2]$ com a segunda palavra de destino recebida e XOR de $S_1[b_3]$ com a terceira palavra de destino recebida. Finalmente, é efetuada a rotação da palavra origem de 24 posições à direita. Estas operações podem ser vistas na figura 2.10.

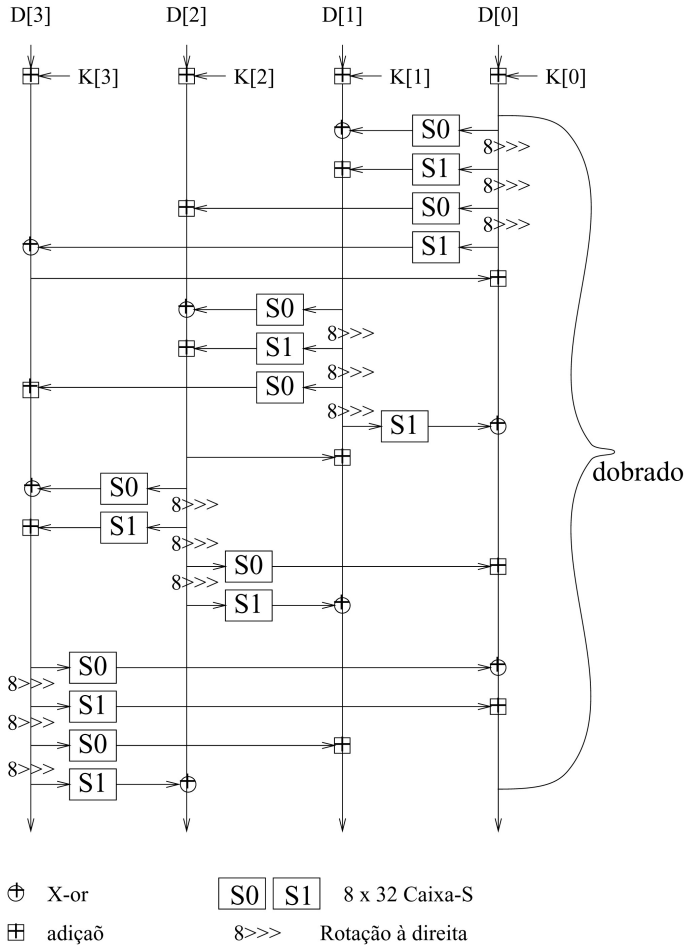

Figura 2.10: Estrutura da Fase Mistura Progressiva: As chaves são adicionadas aos dados, a chave é divida em 4 palavras, cada palavra passa pelas operações de XOR, Adição, Rotação à direita e Caixas-S em conjunto com outra palavra [BUR 99].

Núcleo de Criptografia: Esta fase do algoritmo é composta de 16 rodadas da rede de Feistel tipo 3, sendo 8 rodadas de transformação no modo progressivo e 8 rodadas de transformação no modo regressivo. Esta fase é considerada o coração do MARS. Em cada rodada é utilizada uma função de chaveamento E. Na figura 2.12, podemos ver o diagrama dos modos progressivo e regressivo. Esta função E pega como entrada uma palavra dada e mais duas chaves para retornar três palavras como saída.

O diagrama da função E pode ser visto na figura 2.11.

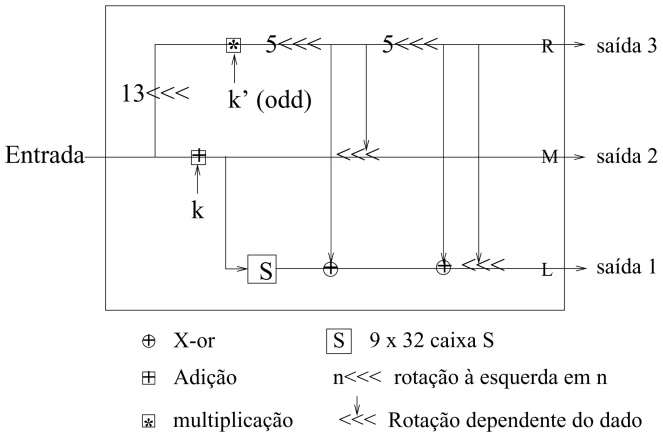

Figura 2.11: Estrutura da Função E: A função recebe como entrada uma palavra de 32 bits que passa pelas operações de XOR, Adição, Multiplicação, Rotação à esquerda, Rotação dependente do dado e Caixas-S, gerando como saída 3 palavras [BUR 99].

Mistura Regressiva: Esta fase é a mesma fase mistura progressiva utilizada para descifrar, exceto que os dados recebidos são processados em diferente ordem, se forem processadas na mesma ordem, a fase mistura progressiva e mistura regressiva se anulam. Em cada rodada, o dado recebido (uma palavra de 32 bits, conhecida como origem) é utilizado para modificar os outros três dados recebidos (conhecidos como palavras destino). Os quatro bytes da palavra origem b_0,b_1,b_2,b_3 são utilizados como índices para as Caixas-S S_0 e S_1, da seguinte forma: b_0,b_2 são os índices para a Caixa-S S_1 e b_1,b_3 são os índices para a Caixa-S S_0. Primeiramente, ocorrem as seguintes operações: XOR de $S_1[b_0]$ com a primeira palavra de destino recebida, subtrai $S_0[b_3]$ da segunda palavra de destino recebida, subtrai de $S_1[b_2]$ da terceira palavra de destino recebida e XOR de $S_0[b_1]$ com a terceira palavra de destino recebida. Finalmente, é efetuada a rotação da palavra origem de 24 posições à direita. Estas operações podem ser vistas na figura 2.13.

No algoritmo MARS, a operação de descifrar é a operação inversa da cifragem, sendo o código de descifrar similar ao código de cifrar, o pseudocódigo do

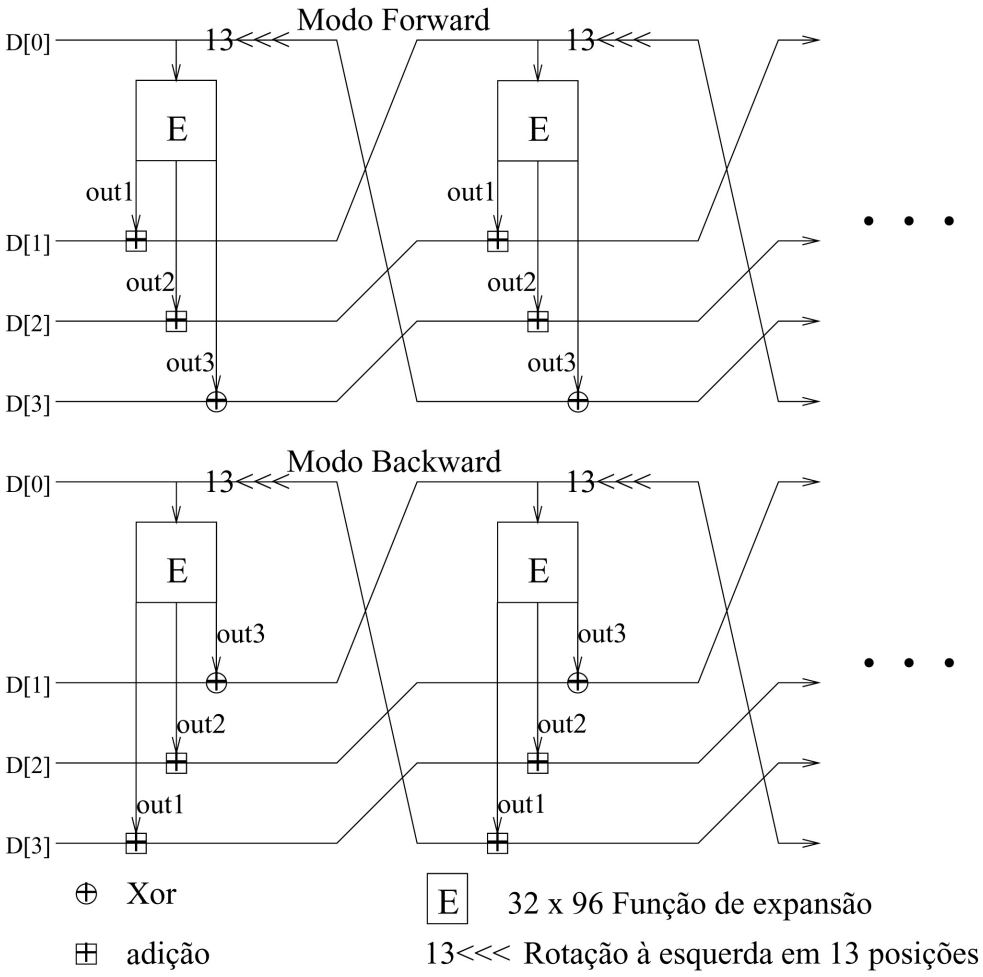

Figura 2.12: Modo Progressivo e Regressivo: É verificado a integração destes modos com a Função E e suas operações de XOR, Adição, Função de Expansão e Rotação à esquerda. Após estes modos são gerados 4 palavras de 32 bits [BUR 99].

algoritmo é representado desta forma [COP 99]:

// Mistura Progressiva

$(A\,,B\,,C\,,D)\;=\;(A\,,B\,,C\,,D)\;(K[0]\,,K[1]\,,K[2]\,,K[3])$

$for\, i\;=\;0\;to\;7\;do\{$

$B\;=\;(B \bigoplus S_0[A]) + S_1[A>>>8]$

$C\;=\;(C + S_0[A>>>16]$

$D\;=\;(D \bigoplus S_1[A>>>24]$

$A\;=\;(A\;>>>\;24)\;+\;B(if\;\;i\;=\;1,5)\;+\;D(if\;\;i\;=\;0,4)$

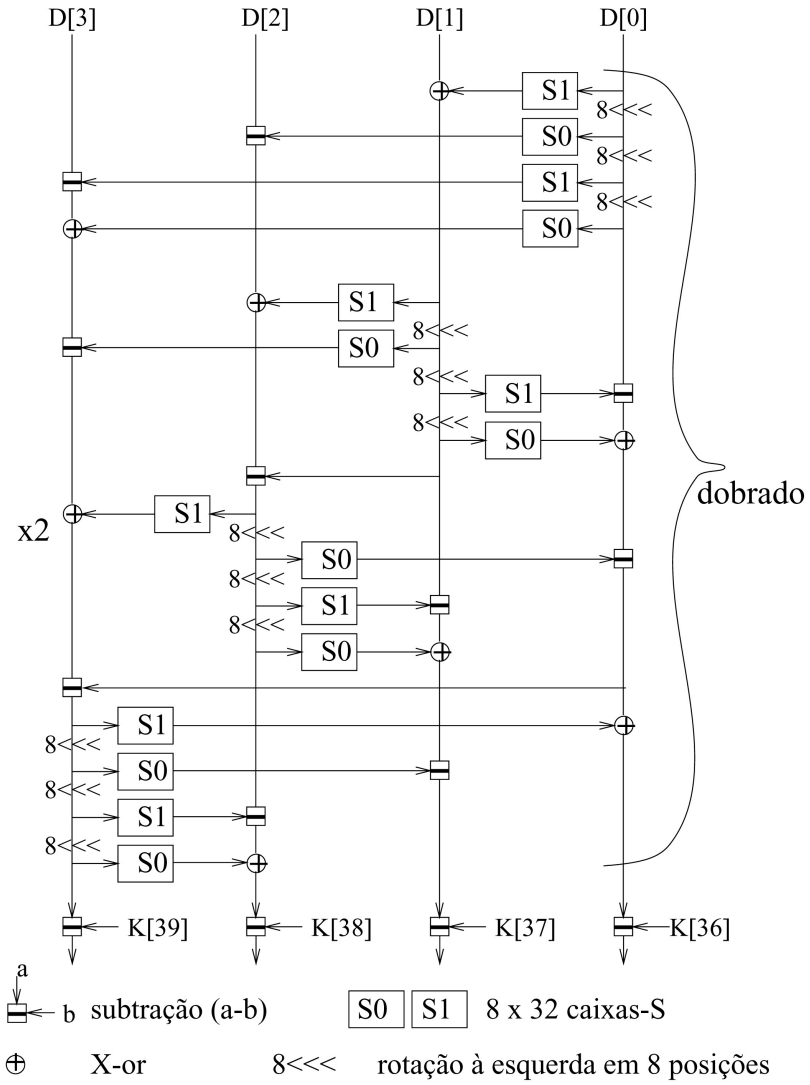

Figura 2.13: Estrutura da Fase Mistura Regressiva: São recebidos como entrada 4 palavras, cada palavra passa pelas operações de XOR, Subtração, Rotação à esquerda e Caixas-S em conjunto com outra palavra, fornecedo como saída 4 palavras de 32 bits [BUR 99].

$$(A\,,B\,,C\,,D)\;-\;(B\,,C\,,D\,,A)$$

}

Núcleo de Criptografia

$for\, i\, =\, 0\ to\ 15\ do\,\{$

$R\, =\, ((A <<< 13)\, X\, K[2_i|5])\ <<<< 10$

$M\, =\, (A\, +\, K[2_i + 4])\ <<<<\ (low\ 5\ bits\ of\ (R >>> 5))$

$L\, =\, (S\,[M] \bigoplus (R >>> 5) \bigoplus R)\ <<<\ (low\ 5\ bits\ of\ R)$

$B\, =\, (B\, +\, L(if\ i\, <\, 8) \bigoplus R(if\ i\, >=\, 8)$

$C\, =\, (C\, +\, M)$

$D\, =\, (D \bigoplus R(if\ i\, <\, 8) + L(if\ i\, >=\, 8)$

$(A\,, B\,, C\,, D)\, =\, (B\,, C\,, D\,, A\ <<<\ 13)$

$\}$

// Mistura Regressiva

$for\, i\, =\, 0\ to\ 7\ do\,\{$

$A\, =\, A\, -\, B(if\ i\, =\, 3, 7)\, -\, D(if\ i\, =\, 2, 6)$

$B\, =\, B \bigoplus S_1[A]$

$C\, =\, C - S_0[A <<< 8]$

$D\, =\, (D - S_1[A <<< 16]) \bigoplus S_0[A\ <<<\ 24]$

$(A\,, B\,, C\,, D)\, =\, (B\,, C\,, D\,, A\ <<<\ 24)$

$\}$

$(A\,, B\,, C\,, D)\, -\, (A\,, B\,, C\,, D)\, -\, (K[36]\,, K[37]\,, K[38]\,, K[39])$

2.6.4 SERPENT

O Serpent foi um projeto apresentado como candidato ao novo AES, ficando entre os cinco algoritmos finalistas [NIS 00], sendo desenvolvido por Ross Anderson, Eli Biham e Lars Knudsen.

O Serpent é um algoritmo que foi projetado mantendo muitas características já pesquisadas e consolidadas no campo dos cifradores simétricos, com o objetivo de manter o que foi pesquisado, bem como assegurar o entendimento do algoritmo de uma forma mais simples. É baseado na estrutura clássica de Feistel, opera com blocos de 128 bits, produzindo um texto cifrado de 128 bits. Permite a utilização de chaves de (128, 192 e 256 bit). Utiliza-se de Caixas-S baseadas na Caixa-S do DES [AND 98].

O algoritmo Serpent é uma SPN de 32 rodadas, operando com quatro palavras de 32 bits, fazendo assim o bloco de 128 bits. O algoritmo cifra um texto aberto de 128 bits para um texto cifrado de 128 bits, utilizando-se de 33 sub-chaves ($\hat{\mathrm{K}}_0, ..., \hat{\mathrm{K}}_{32}$) de 128 bits.

O cifrador é composto das seguintes etapas:

Permutação Inicial (IP): Esta etapa não apresenta nenhuma característica criptográfica significante, foi implementada somente para facilitar na otimização do cifrador. A IP, uma matriz de 8×16, é aplicada sobre o texto aberto P, gerando $\hat{\mathrm{B}}_0$ que serve como entrada para a primeira rodada.

32 Rodadas: Cada rodada $i \, \epsilon \, \{0, \, ... \, , 31\}$, consiste de apenas três operações.

1 - Mistura de Chaves: A cada rodada, uma subchave K_i de 128 bits sofre um XOR com o valor intermediário de B_i.

2 - Caixas-S: A combinação dos 128 bits e a chave são consideradas as quatro palavras de 32 bits de entrada. A Caixa-S é aplicada sobre estas quatro palavras, gerando como saída também quatro palavras. Cada rodada R_i $(i \, \epsilon \, \{0, \, ... \, , 31\})$ utiliza uma das oito Caixas-S que foram replicadas e que são utilizadas quatro vezes em todas as 32 rodadas. Isto ocorre da seguinte forma: R_0 usa S_0, a primeira cópia de S_0 utiliza os bits $0, \, 1, \, 2, \, 3$ de $\hat{\mathrm{B}}_0 \bigoplus \hat{\mathrm{K}}_0$ como entrada e retorna como saída os primeiros quatro bits de um vetor intermediário, a próxima cópia de S_0 utiliza como entrada os bits de 4 a 7 de $\hat{\mathrm{B}}_0 \bigoplus \hat{\mathrm{K}}_0$ e retorna como saída os próximos quatro bits de um vetor intermediário, e assim sucessivamente. Este vetor intermediário será utilizado pela transformação linear que irá ocorrer no próximo passo. A última rodada R_{31} é diferente das outras, nela é aplicado S_7 sobre $\hat{\mathrm{B}}_{31} \bigoplus \hat{\mathrm{K}}_{31}$, o resultado será feito um XOR com $\hat{\mathrm{K}}_{32}$ antes de ter passado pela transformação linear [$\hat{\mathrm{B}}_{32} := S_7 \, (B_{31} \bigoplus K_{31}) \bigoplus K_{32}$], o resultado $\hat{\mathrm{B}}_{32}$ sofrerá a FP e gerará o texto cifrado.

3 - Transformação Linear: Os 32 bits de cada saída são transformados linearmente, a utilização desta transformação ocorreu em função de maximizar-se o

efeito avalanche, isto ocorre da seguinte forma:

$$X_0, X_1, X_2, X_3 := S_i\,(B_i \bigoplus K_i)$$
$$X_0 := X_0 <<<\ 13$$
$$X_2 := X_2 <<<\ 3$$
$$X_1 := X_1 \bigoplus X_0 \bigoplus X_2$$
$$X_3 := X_3 \bigoplus X_2 \bigoplus (X_0 <<\ 3)$$
$$X_1 := X_1 <<<\ 1$$
$$X_3 := X_3 <<<\ 7$$
$$X_0 := X_0 \bigoplus X_1 \bigoplus X_3$$
$$X_2 := X_2 \bigoplus X_3 \bigoplus (X_1 <<\ 7)$$
$$X_0 := X_0 <<<\ 5$$
$$X_2 := X_2 <<<\ 22$$
$$B_{i+1} := X_0, X_1, X_2, X_3$$

onde $<<<$ significa rotação à esquerda e $<<$ significa deslocamento à esquerda.

Permutação Final (FP): Esta etapa também não apresenta nenhuma característica criptográfica significante, foi implementada somente para facilitar a otimização do cifrador. A FP também é uma matriz de 8×16. A FP é aplicada sobre o $\hat{\mathrm{B}}_{32}$, gerando texto cifrado C.

O algoritmo pode ser descrito formalmente da seguinte forma:

$$\hat{\mathrm{B}}_0 := IP\,(P)$$
$$\hat{\mathrm{B}}_{i+1} := R_i\,(\hat{\mathrm{B}}_i)$$
$$C := FP\,(\hat{\mathrm{B}}_{32})$$

onde:

$$R_i(X) = L(\hat{\mathrm{S}}_i(X \bigoplus \hat{\mathrm{K}}_i))\ com\ i\ =\ 0, ..., 30$$
$$R_i(X) = \hat{\mathrm{S}}_i(X \bigoplus \hat{\mathrm{K}}_i) \bigoplus \hat{\mathrm{K}}_{32}\ com\ i\ =\ 31$$

O processo de cifrar e descifrar são diferentes no Serpent, pois uma Caixa-S inversa deve ser usada em reversa ordem, bem como a transformação linear e as subchaves.

2.6.5 TWOFISH

O Twofish foi projetado por Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall e Niels Ferguson. Candidato ao novo padrão de criptografia coordenado pelo NIST [NIS 00], chegando até a segunda rodada ficando entre os cinco finalistas.

Twofish é um cifrador de blocos de 128 bits, com chaves variáveis (128, 192 e 256 bits), podendo aceitar chaves maiores que 256 bits. É uma estrutura clássica de Feistel de 16 rodadas com uma função bijetiva F, esta sempre não linear [SCH 98].

As Caixas-S dão condição de não linearidade ao algoritmo, o Twofish se utiliza de quatro diferentes Caixas-S, bijetivas, dependentes da chave, estas são Caixas-S de 8 por 8 bits.

Conforme a ilustração da figura 2.14, podemos verificar o algoritmo Twofish.

O processo de cifrar do Twofish, apresentado na figura 2.14, ocorre utilizando os seguintes passos:

- O texto aberto de 128 bits é fornecido como entrada.
- O texto aberto é dividido em quatro palavras de 32 bits.
- Com cada palavra de entrada é efetuado um XOR especial com quatro palavras da chave (K_0,K_1,K_2 e K_3). Este XOR especial é efetuado utilizando uma técnica inventada por Rivest para o DES-X, conhecida como *Whitening*. Após o *Whitening*, 16 rodadas são efetuadas.
- Em cada rodada, as duas palavras da esquerda irão servir como entrada para a função F.

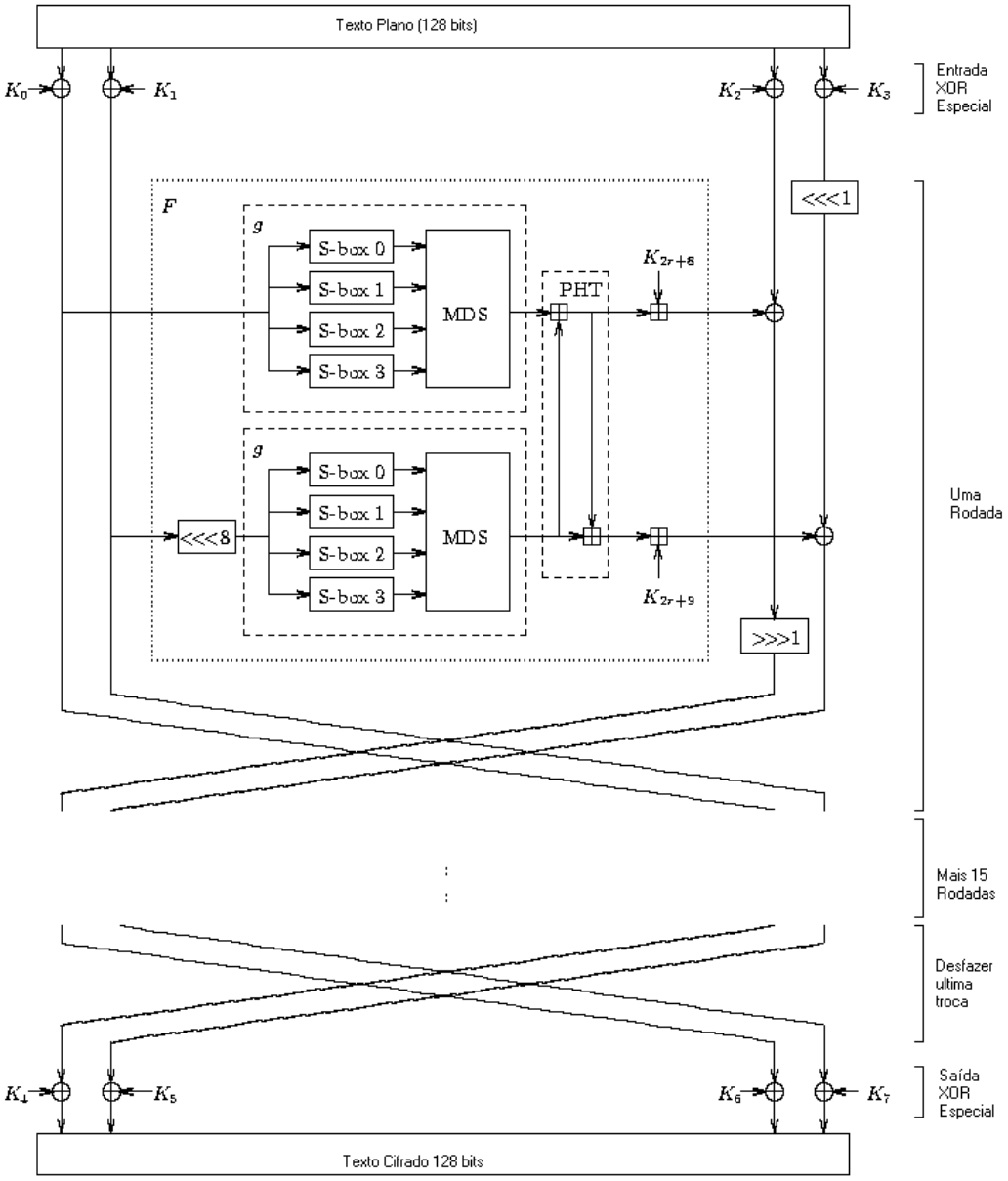

Figura 2.14: Cifrador de Blocos Twofish: O algoritmo é apresentado com suas 16 rodadas, as suas operações e sua função F. O algoritmo recebe como entrada um texto aberto de 128 bits e fornece um texto cifrado de 128 bits como saída [SCH 98].

- As duas palavras de saída que passaram pela função F irão sofrer um XOR com as palavras da direita, sendo que uma delas sofrerá uma rotação à esquerda de 1 bit antes do XOR e outra uma rotação à direita de 1 bit após o XOR.

- A parte da esquerda e da direita são trocadas de lado, servindo como entrada para a próxima rodada.
- Após todas as 16 rodadas, é desfeito a última troca de lado da parte esquerda com a direita.
- As quatro palavras de saída sofrem o *Whitening* com mais quatro palavras da chave (K_4,K_5,K_6 e K_7).
- O texto cifrado de 128 bits é fornecido como saída.

A função F é uma permutação dependente da chave e opera com 64 bits. Os passos efetuados pela função F são:

- A função F recebe três argumentos, duas palavras de entrada R_0 e R_1 e o número da rodada r, que é usado para selecionar a subchave adequada.
- R_0 serve como entrada para função g, gerando T_0. Sendo, $T_0 = g\,(R_0)$.
- R_1 é rotacionado 8 bits à esquerda e depois serve como entrada para função g, gerando T_1. Sendo, $T_1 = g\,(ROL\,(R_1,8))$, onde ROL é a função que efetua a rotação à esquerda do primeiro elemento em função dos bits do segundo argumento.
- Os resultados T_0 e T_1 são combinados em uma PHT (Pseudo-Hadamard Transformada) e duas palavras da chave expandida são adicionadas. Gerando $F_0 = (T_0 + T_1 + K_{2r+8})\ mod\ 2^{32}$ e $F_1 = (T_0 + 2T_1 + K_{2r+9})\ mod\ 2^{32}$.
- O resultado da função F é $(F_0\,, F_1)$.

A função g é considerada o coração do Twofish. Os passos efetuados pela função g são:

- A entrada é dividida em quatro bytes.
- Cada byte é processado pela sua própria Caixa-S dependente da chave. Estas Caixas-S são bijetivas e pegam 8 bits de entrada e produzem 8 bits de saída.

- Os quatro bytes resultantes das Caixas-S são utilizados como um vetor de tamanho quatro sobre $GF^4(2^8)$ e multiplicado pela matriz MDS (Máxima Distância Separável) de 4 $\times$ 4 (usando o campo $GF(2^8)$ para os cálculos).

- O vetor resultante desta operação é a palavra de 32 bit que é o resultado de g.

A segurança do cifrador está muito relacionada a suas Caixas-S. No caso do Twofish, este se utiliza da mesma solução adotada pelo cifrador Square, Caixas-S pequenas (8 por 8 bits), que são utilizadas para construir Caixas-S médias (8 por 32 bits) [SCH 98].

Cada Caixa-S é definida por dois, três ou quatro bytes da chave. Esta quantidade depende do tamanho da chave adotado pelo Twofish.

2.6.6 RIJNDAEL

O Rijndael foi o projeto apresentado como candidato ao novo AES, sendo o grande vencedor do evento [NIS 00], desenvolvido por Joan Daemen e Vicent Rijmen.

O Rijndael é um cifrador de blocos interativo que opera com tamanhos de blocos e chaves variáveis, podendo seu tamanho ser de 128, 192 e 256 bits. O tamanho do bloco e a chave podem ser especificados independentemente um do outro. O algoritmo não apresenta, em seu projeto, a tradicional estrutura de Feistel [DAE 99].

No lugar da tradicional estrutura de Feistel, foi colocada uma estrutura de camadas (transformações inversíveis e uniformes), esta escolha aconteceu em função deste método prover resistência contra a criptoanálise linear e diferencial. A quantidade de rodadas do algoritmo é dependente do tamanho do bloco e do tamanho da chave utilizados [DAE 99].

Cada camada possui sua própria rodada de transformação, que é composta da seguinte forma:

[4]GF = Galois Field - Corpo de Galois

Camada de transformação linear: É responsável por garantir grande difusão às múltiplas rodadas.

Camada não linear: Utilização de Caixas-S que fornecem ótima propriedade não linear.

Camada de adição de chaves: Aplicação de um simples XOR sobre a chave da rodada.

O projeto do cifrador é bem simples e pode ser representado pelos seguintes passos:

$1-$ **Uma rodada inicial de adição de chaves.**

$2-$ $(N-1)$ **rodadas de transformação.**

$3-$ **Rodada final** N**.**

O pseudocódigo do algoritmo em $Linguagem\ C$ pode ser apresentado da seguinte forma [DAE 99]:

$Rijndael\,(Estado,\, ChaveCifrador)$
{
$ExpansaoChaves\,(ChaveCifrador,\, ChaveExpandida);$
$RodadaAdicaoChaves\,(Estado,\, ChaveExpandida);$
$For\,(i = 1;\, i < Nr;\, i++)\ \ Rodada\,(Estado,\, ChaveExpandida + Nb * i);$
$RodadaFinal\,(Estado,\, ChaveExpandida + Nb * Nr);$
}

A função $Rodada$ é definida da seguinte forma [DAE 99]:

$Rodada\,(Estado,\, ChaveRodada)$
{
$SubstituicaoByte\,(Estado);$
$LinhaRotacao\,(Estado);$
$MisturaColunas\,(Estado);$

$RodadaAdicaoChaves\,(Estado,\, ChaveRodada)$;

}

A função $RodadaFinal$ é um pouco diferente, é definida da seguinte forma [DAE 99]:

$RodadaFinalRound\,(Estado,\, ChaveRodada)$

{

$SubstituicaoByte\,(Estado)$;

$LinhaRotacao\,(Estado)$;

$RodadaAdicaoChaves\,(Estado,\, ChaveRodada)$;

}

As funções utilizadas pela rodada de transformação operam da seguinte forma:

Transformação SubstituicaoByte: É a transformação não linear realizada pela Caixa-S, esta Caixa-S é inversível e é construída pela composição de duas transformações (Multiplicativa inversa em $GF(2^8)$ e transformação afim). Esta Caixa-S é aplicada sobre todos os bytes intermediários conhecidos como $State$. A figura 2.15, ilustra esta operação.

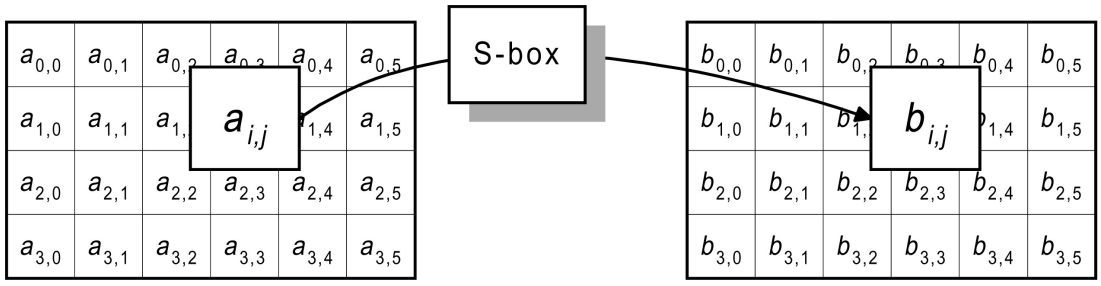

Figura 2.15: Operação SubstituicaoByte do Rijndael: Operação não linear do algoritmo, um byte da tabela State é substituído por outro [DAE 99].

Transformação LinhaRotacao: É a rotação cíclica efetuada sobre a linha do $State$, em

função dos parâmetros $C1,\ C2\,e\,C3$, sendo, $C1,\ C2\,e\,C3$ dependentes do tamanho do bloco, podendo variar de $1\,a\,4$. A figura 2.16, abaixo, ilustra esta operação.

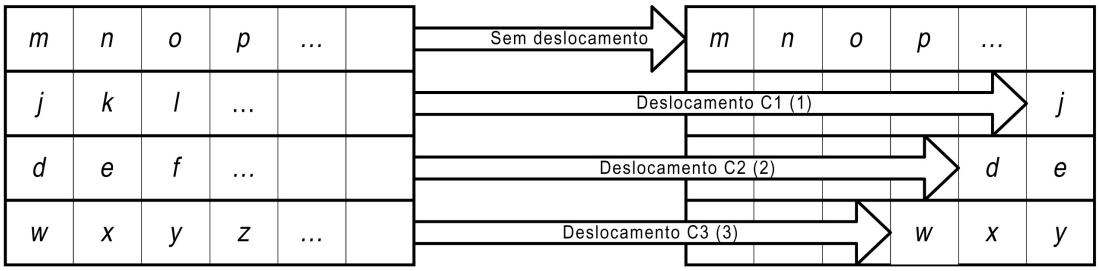

Figura 2.16: Operação LinhaRotação do Rijndael: A operação de rotação dependente do tamanho do bloco [DAE 99] .

Transformação MisturaColunas: Esta operação de permutação é efetuada sobre as colunas do $State$, estas servem para definir um polinômio sobre $GF(2^8)$, que é multiplicado por módulo $X_4 + 1$ do polinômio $c(x)$ dado. A figura 2.17, ilustra esta operação de permutação.

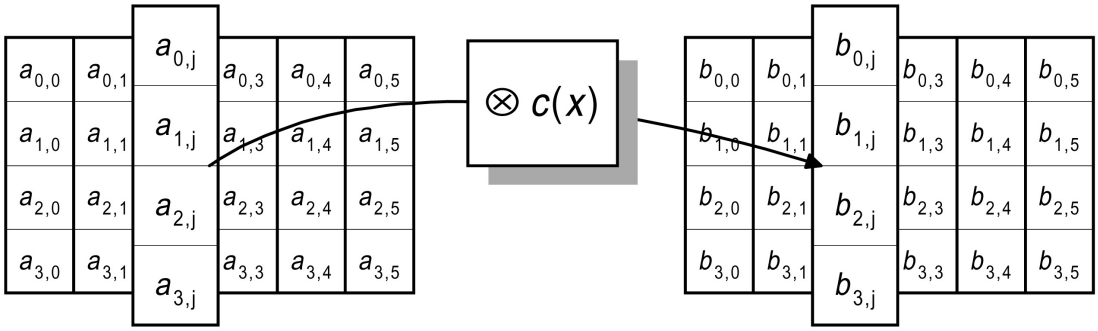

Figura 2.17: Operação MituraColunas do Rijndael: Operação de permutação do algoritmo onde uma coluna da tabela State é permutada por outra [DAE 99].

Rodada de Adição de Chaves: Nesta operação, a chave da rodada sofre um simples XOR com o $State$. A chave da rodada é derivada da chave do cifrador, o tamanho

da chave da rodada é igual ao tamanho do bloco usado. A figura 2.18, ilustra esta operação.

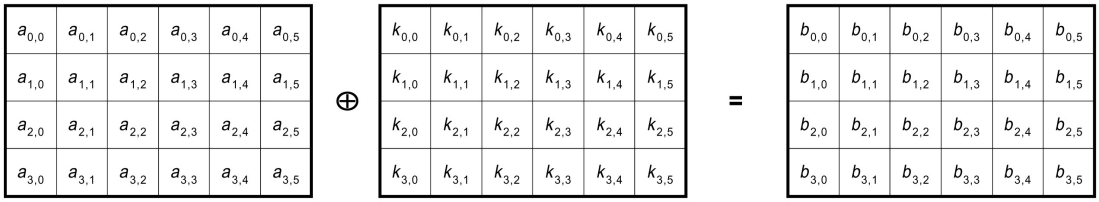

Figura 2.18: Operação Rodada Adição de Chaves do Rijndael: A chave da rodada sofre a operação XOR com a tabela State [DAE 99].

Para descifrar é necessário efetuar a inversa das transformações e implementar em um algoritmo diferente do utilizado para cifrar, portanto, o algoritmo não é simétrico.

2.7 Comparação entre os cifradores

Na tabela 2.9, iremos apresentar um resumo contendo as principais características dos cifradores analisados anteriormente.

Tabela 2.9: Principais Características dos Cifradores Analisados

Cifrador	**Estrutura**	**Caixa-S**	**Bloco**	**Chave**	**Reversível**
DES	Feistel Completa	Sim e Fixa	64 Bits	56 Bits	Sim
CAST-256	Feistel Incompleta	Sim e Fixa	128 Bits	256 Bits	Sim
TWOFISH	Feistel Completa	Sim e Variável	128 Bits	256 Bits	Não
MARS	Feistel Completa	Sim e Fixa	128 Bits	Variável	Sim
SERPENT	Feistel Completa	Sim e Fixa	128 Bits	Variável	Sim
RIJNDAEL	Não é Feistel	Sim e Fixa	Variável	Variável	Não

Com relação as principais características dos cifradores analisados na tabela 2.9, podemos tirar as seguintes conclusões:

Estrutura: Dos cifradores analisados a estrutura de Feistel Completa é apresentada no projeto dos cifradores DES, TWOFISH, MARS e SERPENT. A estrutura de Feistel

Incompleta foi apresentada somente no projeto do CAST-256. O cifrador RIJNDAEL não apresenta a estrutura de Feistel em seu projeto. Podemos verificar pelos cifradores analisados que a estrutura de Feistel completa é muito utilizada no projeto de cifradores de bloco.

Caixas-S: Todos os cifradores analisados com exceção do TWOFISH apresentam em seu projeto as Caixas-S de forma fixa, ou seja, o conteúdo da Caixa-S é sempre o mesmo. A Caixa-S do TWOFISH não é fixa pois é gerada usando elementos da chave privada utilizada.

Bloco: O tamanho do bloco se refere a quantidade de bits que serão operados pelo cifrador. Os cifradores CAST-256, TWOFISH, MARS e SERPENT operam com blocos de 128 bits. O cifrador DES opera com blocos de 64 bits. O cifrador RIJNDAEL opera com blocos de tamanho variável (64 bits, 128 bits e 256 bits).

Tamanho da Chave: O tamanho da chave se refere a quantidade de bits que compõe a chave utilizada pelo cifrador. Os cifradores CAST-256 e TWOFISH utilizam somente chaves de 256 bits. O DES utiliza chaves de somente 56 bits. Os cifradores MARS, SERPENT e RIJNDAEL podem utilizar chaves de diversos tamanhos. O tamanho da chave é considerado um dos fatores importantes para a segurança dos algoritmos de criptografia.

Reversível: A reversibilidade é a capacidade do mesmo algoritmo ser utilizado tanto no processo de cifrar como no de descifrar. Dos cifradores analisados o DES, CAST-256, MARS e SERPENT são reversíveis e os cifradores TWOFISH e RIJNDAEL não são reversíveis.

2.8 Conclusão

Este capítulo apresentou os conceitos de Criptografia, tratou de descrever os principais cifradores de bloco que apresentam em sua estrutura Caixas-S, cifradores

mais tradicionais como o DES e os cifradores candidatos ao AES, como o CAST, MARS, SERPENT, TWOFISH, incluindo o AES (RIJNDAEL).

Com base no estudo dos principais cifradores de bloco analisados, foi elaborada uma tabela que compara suas principais características, que são: Tipode de Estrutura; Tipo de Caixa-S; Tamanho do Bloco; Tamanho da Chave; Reversibilidade.

Desta tabela, pudemos verificar que somente o cifrador RIJNDAEL não adotou em seu projeto a estrutura de Feistel, estrutura que tem sido muito analisada e estudada, mas mesmo assim, o seu projeto foi considerado o mais adequado pelo AES.

Capítulo 3

Caixas-S

3.1 Introdução

Este capítulo tem como objetivo a apresentação do histórico das estruturas de substituição S (Caixas-S), suas principais características, assim como apresentar critérios de avaliação e projeto de Caixas-S usadas pelos cifradores DES, CAST, MARS, SERPENT, TWOFISH e RIJNDAEL e realizar um comparativo destas características.

Verifica-se na literatura um grande esforço no estudo de como projetar e avaliar Caixas-S [SCH 96, STA 99].

Trabalhos em nível de mestrado e doutorado têm sido apresentados sobre o assunto. Dentre os mais completos trabalhos sobre Caixas-S, podemos destacar os seguintes:

a) Trabalho em nível de mestrado apresentado por Liam Keliher à Universidade de Queen's no Canadá, onde faz um estudo de Caixas-S dependentes da chave [KEL 97].

b) Trabalho em nível de doutorado apresentado por Amr Mohamed Youssef à Universidade de Queen's no Canadá, onde faz um estudo das propriedades dos mapeamentos das funções booleanas e apresenta um novo método para construir Caixas-S com alto grau de não linearidade. Também apresenta a formalização matemática dos principais critérios a serem adotados em projetos de Caixas-S [YOU 97].

c) Trabalho em nível de doutorado apresentado por Kwangjo Kim à Universidade Nacional de Yokohama no Japão, onde faz um estudo das propriedades das funções booleanas que satisfazem aos critérios do SAC e a relação existente entre as funções Bent e booleanas que satisfazem os critérios do SAC. Utilizando estas características, apresenta um novo conjunto de Caixas-S que podem ser usadas em substituição as Caixas-S do cifrador DES [KIM 90].

Não podemos deixar de destacar a contribuição do pesquisador Stafford E. Tavares e seus orientados da Universidade de Queen's, no Canadá, para esta área de pesquisa, com inúmeros trabalhos publicados.

Começamos fazendo um histórico das Caixas-S. Em seguida apresentamos suas principais características, as formas de projetar projetar Caixas-S e critérios de projetos de Caixas-S. Apresentamos também uma comparação contendo as principais características dos algoritmos analisados com relação as suas Caixas-S.

3.2 Histórico

Os cifradores de bloco simétricos, em sua maioria, são baseados nas idéias de Shannon sobre difusão e confusão. A difusão normalmente é produzida por uma estrutura de substituição conhecida por caixa de substituição S ou simplesmente Caixa-S.

Como estas estruturas são responsáveis pela segurança do algoritmo, um considerável esforço vem sendo realizado para analisar o projeto e as operações das Caixas-S. Esta análise consiste em verificar quão melhor é uma Caixa-S em relação a outra.

Nas tabelas 3.1 e 3.2 abaixo, encontram-se as principais pesquisas de projeto e avaliação de Caixas-S:

Maiores detalhes sobre estas contribuições podem ser encontradas na Home Page do pesquisador Terry Riter, que apresentou este resumo histórico sobre o desenvolvimento do projeto de Caixas-S [RIT 99].

Tabela 3.1: Histórico de pesquisas de projeto e avaliação de Caixas-S

1973	Horst Feistel	Introduziu o conceito de *Efeito Avalanche*.
1979	Kam e Davida	Introduziram o conceito de *Completeza* [1]
1982	Gordon e Retkin	Apresentaram um trabalho sobre Caixas-S que contém relacionamentos lineares.
1982	Ayoub	Sugere uma técnica para analisar quais cifradores baseados em substituições e permutações estão livres de vulnerabilidades.
1985	Webster e Tavares	Revisaram os conceitos de Completeza e Avalanche, criando um novo conceito o SAC - [2] Critério Rigoroso de Avalanche.
1988	Pieprzyk e Finkelstei	Apresentaram um trabalho sobre a esperada não linearidade das Caixas-S escolhidas de forma aleatória.
1988	Rejanè Forré	Apresentou a aplicação de testes de SAC sobre funções Walsh.
1989	Meier e Staffelbach	Apresentaram um trabalho sobre não linearidade ”perfeita” e relacionaram este aspecto com a difusão do SAC.
1989	Pieprzyk	Apresentou indicadores da propriedade de propagação de erros de funções booleanas, utilizados pelo SAC.
1989	Pieprzyk e Finkelstein	Trabalho consagrado por projetar e construir permutações booleanas não lineares (Caixas-S).
1989	Pieprzyk	Discutiu a não linearidade das permutações.
1990	Lloyd	Efetuou a investigação da conexão entre as três propriedades de uma função binária.
1990	Preneel e outros	Efetuaram a generalização do SAC e do critério da não linearidade perfeita para o Critério de Propagação de ordem k.
1991	Nyberg	Apresentou a construção de Caixas-S não-linear perfeita.

3.3 Características das Caixas-S

As Caixas-S são estruturas muito simples e apresentam as seguintes características:

- É uma matriz bidimensional com x linhas e y colunas.
- Efetua o mapeamento de n bits de entrada em m bits de saída. Em função desta característica, quando nos referimos a uma Caixa-S, dizemos que ela é do tipo n $\times$ m.
- É uma estrutura somente de substituição.
- Geralmente, é a única operação não linear do algoritmo de criptografia.

Tabela 3.2: Histórico de pesquisas de projeto e avaliação de Caixas-S. Continuação da Tabela 3.1

1991	Dawson e Tavares	Apresentaram um conjunto de novos critérios para o projeto de Caixas-S baseado na teoria da informação.
1992	Sivabalan, Tavares e Peppard	Discutiram a questão de vazamento de informações em Caixas-S e projeto de cifradores de substituição e permutação.
1992	Adams	Propôs a utilização de funções *Bent* no projeto de Caixas-S.
1993	Cusick	Apresentou um trabalho em que realiza a contagem do número de funções que satisfazem o SAC de ordem $n-4$.
1993	O'Connor	Efetuou o estudo dos efeitos da criptanálise diferencial na seleção de Caixas-S de forma aleatória.
1994	Daemen, Govaerts and Vandewalle	Introduziram o conceito de correlação de matrizes de mapeamentos booleanos.
1995	Youssef, Tavares, Mister e Adams	Apresentaram uma forma de estimar a não linearidade esperada da seleção aleatória de Caixas-S.
1995	Youssef e Tavares	Apresentaram um discussão referente à imunidade da seleção aleatória de Caixas-S com relação a cripoanálise diferêncial e linear.
1995	Youssef e Tavares	Apresentaram a probabilidade de se escolher uma Caixa-S em que as cordenadas de saída sejam uma função afin.
1995	Youssef e Tavares	Discutiram o vazamento de informações quando se escolhe funções aleatoriamente.
1995	Zhang e Zheng	Revisaram o SAC e introduziram o conceito de GAC [3] - Característica Global de Avalanche.
1995	Vaudenay	Apresentou um trabalho tentando provar que as propriedades lineares não são tão importantes no projeto de Caixas-S.

- O seu tamanho é muito importante para a segurança do algoritmo, Caixas-S grandes são mais resistentes às técnicas de criptoanálise diferencial e linear, mas, em contrapartida, existe uma complexidade maior no seu projeto e na disposição das mesmas [STA 99].

- O tamanho de m é mais importante que n, pois aumentando o tamanho de n reduz a eficácia da criptoanálise diferencial, mas grandes aumentos acrescem a eficácia da criptoanálise linear [SCH 96].

- Quando $n = m$ o mapeamento é reversível e é chamado de bijetiva.

- Existem diferentes técnicas para se projetar uma Caixa-S [ROB 95].

3.4 Projeto de Caixas-S

Nyberg, em suas pesquisas, identificou quatro maneiras de projetar Caixas-S [ROB 95], sendo elas:

Randômica: A escolha é feita aleatoriamente, sem satisfazer nenhum critério.

Randômica com Testes: A escolha é feita aleatoriamente e esta escolha deve atender a certos critérios.

Manual: Técnica que se baseia na utilização de matemática elementar, conduzindo-se de forma mais intuitiva. Aparentemente esta é a técnica escolhida para o projeto das Caixas-S do DES.

Matemática: Geração das Caixas-S baseada, em princípios matemáticos. Esta técnica permite que o projeto de Caixas-S seja mais seguro contra a criptoanálise diferencial e linear. Esta técnica é utilizada pelo cifrador CAST.

Uma variação, utilizando a primeira técnica, consiste na geração das Caixas-S aleatoriamente em função da chave usada. Um estudo mais aprofundado sobre este assunto encontra-se em Keliher [KEL 97]. Esta técnica é utilizada pelo algoritmo Blowfish . Uma grande vantagem desta abordagem é a sua dependência da chave, isto é: como a chave não é fixa, fica impossível a análise das Caixas-S buscando vulnerabilidades [STA 99]. Uma grande desvantagem desta abordagem é a diminuição da performance do algoritmo em função da necessidade da realização desta operação.

Abaixo, iremos apresentar as principais técnicas utilizadas na construção de Caixas-S dos cifradores, estudados no capítulo 2.5.

DES: Utilizam-se oito pequenas Caixas-S de 4 linhas por 16 colunas com 4 bits cada. Em cada Caixa-S, entram 6 bits e saem 4 bits, ou seja, 6×4. O seu projeto de Caixa-S foi elaborado de forma manual (matemática elementar), de forma mais intuitiva, maiores detalhes são apresentados em [ROB 95, SCH 96, STA 99].

CAST-256: Utilizam-se quatro Caixas-S médias de 32 linhas por 8 colunas com 32 bits cada. Em cada Caixa-S, entram 8 bits e saem 32 bits, ou seja, 8×32. O seu projeto de Caixa-S foi elaborado visando dificultar a criptoanálise linear e diferencial, as suas Caixas-S foram geradas utilizando-se funções bent parciais, maiores detalhes sobre esta técnica podem ser conseguidos em [ADA 90, KIM 90, ADA 93, MIS 96, YOU 97, STA 99].

TWOFISH: Utilizam-se quatro pequenas Caixas-S de 1 linha por 16 colunas com 4 bits cada, para formar Caixas-S médias de 32×8. Em cada Caixa-S entram 8 bits e saem 8 bits, ou seja, 8×8. O seu projeto de Caixa-S está baseado na utilização da técnica randômica com testes, as Caixas-S foram construídas baseadas em Caixas-S de 4×4 que atendiam aos critérios desejados, até chegar nas Caixas-S de 8×8 desejadas [SCH 98].

MARS: Utiliza-se uma grande Caixa-S de 16 linhas por 32 colunas com 32 bits cada. Nesta Caixa-S entram 8 bits e saem 32 bits, ou seja, 8×32. O seu projeto de Caixa-S está baseado na utilização da técnica randômica com testes, estes testes foram realizados até chegarem em uma Caixa-S que tivesse boas características diferencial e linear, este processo durou uma semana [COP 99, BUR 99].

SERPENT: Utilizam-se oito Caixa-S grandes de 16 linhas por 32 colunas com 32 bits cada. Em cada Caixa-S, entram 4 bits e saem 4 bits, ou seja, 4×4. O seu projeto de Caixas-S foi inspirado no RC4, que utiliza uma matriz de 32 linhas com 16 entradas, as 32 linhas foram preenchidas utilizando as Caixas-S do DES e sofreram transformações até oferecerem as características desejadas (diferencial e linear), isto foi realizado até conseguir oito Caixas-S com estas características. Esta técnica pode ser classificada como manual, maiores detalhes sobre a geração podem ser encontradas em [AND 98].

RIJNDAEL: Utiliza-se uma Caixa-S média de 16 linha por 16 colunas com 8 bits cada. Nesta Caixa-S entram 32 bits e saem 32 bits, ou seja, 32×32. O seu projeto de Caixas-S foi elaborado utilizando manipulações algébricas em $GF(2^8)$ [DAE 99]

com o atendimento aos critérios apresentados por Nyberg em [NYB 94]. Esta técnica pode ser classificada como matemática, apesar de usar mapeamentos algébricos simples.

Na tabela 3.3, encontra-se um resumo das principais características dos projetos de Caixas-S.

Tabela 3.3: Características dos Projetos de Caixas-S Analisadas

Cifrador	**Tamanho da Caixa-S**[4]	**Tipo do Projeto**	**Entra**	**Sai**
DES	Pequenas $4l \; x \; 16c$ de 4 bits	Manual	6 bits	4 bits
CAST-256	Média $32l \; x \; 8c$ de 32 bits	Matemática	8 bits	32 bits
TWOFISH	Pequenas $8l \; x \; 8c$ de 4 bits	Randômica com Testes	8 bits	8 bits
MARS	Grande $16l \; x \; 32c$ de 32 bits	Randômica com Testes	8 bits	32 bits
SERPENT	Grande $32l \; x \; 16c$ de 32 bits	Manual	4 bits	4 bits
RIJNDAEL	Média $16l \; x \; 16c$ de 8 bits	Matemática	32 bits	32 bits

Para a classificação do tipo de projeto foi utilizado o trabalho de Nyberg [ROB 95].

Como relação aos projetos analisados Tabela 3.1, podemos tirar as seguintes conclusões:

Tamanho da Caixa-S: Os projetos modernos utilizam Caixas-S médias e grandes, pois, como foi visto, estas apresentam melhores características de segurança em comparação as pequenas Caixas-S utilizadas pelo DES. Esta particularidade não se aplica ao cifrador TWOFISH que utiliza pequenas Caixas-S, geradas dependentes da chave, o que lhe garante boas características de segurança. As Caixa-S médias apresentam matrizes de 256 elementos de 32 bits possuindo 8.192 bits, as Grandes apresentam matrizes de 512 elementos de 32 bits possuindo 16.384 bits.

Tipo de Projeto: Podemos verificar que os projetos DES e SERPENT foram baseados na técnica Manual por apresentarem utilização de matemática elementar. Os projetos TWOFISH e MARS foram baseados na técnica Randômica com Testes, sendo os projetos CAST-256 e RIJNDAEL os únicos a utilizaram projetos baseados na utilização de princípios matemáticos.

Bits de Entrada: Como podemos verificar, os bits de entrada dos cifradores são de 4 bits, 6 bits e 8 bits com exceção do RIJNDAEL que apresenta 32 bits. Como foi apresentado, é desejável que os bits de entrada não sejam muito grandes, pois aumentam a eficácia da criptoanálise linear e diminuem a eficácia da criptoanálise diferencial. O projeto do RIJNDAEL preferiu aumentar a eficácia da criptoanálise diferencial no seu projeto.

Bits de Saída: Os bits de saída das Caixas-S são mais importantes que os bits de entrada. Desta forma, podemos verificar que três projetos (RIJNDAEL, MARS e CAST-256) apresentaram um número grande de bits de saída 32 bits, os projetos DES e SERPENT apresentam 4 bits de saída e o projeto do TWOFISH apresenta 8 bits de saída.

3.5 Critérios de Projeto de Caixas-S

Para poder avaliar o projeto de uma Caixa-S, foram desenvolvidos critérios. O objetivo desses critérios é verificar se as Caixas-S projetadas atendem a determinadas características desejadas e necessárias para manter a segurança do algoritmo.

Abaixo, iremos apresentar os principais critérios:

Não Linearidade: É a característica em que cada bit do texto cifrado apresenta baixa correlação para com um sistema linear de equações [HEY 93]. Mais detalhes sobre este critério se encontram no capítulo 5.

Efeito Avalanche: Uma função $f : \{0,1\}^t \rightarrow \{0,1\}^t$ atende a este critério, se, em média, metade dos bits de saída são alterados quando um simples bit de entrada sofre alteração [FEI 73].

Completeza: Dada uma correspondência um-para-um $f : \{0,1\}^n \rightarrow \{0,1\}^n$, esta atende a este critério, se para cada i, j de $\{1, ..., n\}$ existem dois vetores de n-bit X_1 e X_2, tal que, X_1 e X_2 diferem somente do i-ésimo bit e $f(X_1)$ difere de $f(X_2)$, pelo

menos do j-ésimo bit, ou seja, cada bit de saída dependente de todos os bits de entrada [KAM 79].

SAC-Critério Rigoroso de Avalanche: O SAC é a combinação dos critérios Efeito Avalanche e Completeza. Para que a função criptográfica atenda a este critério, cada bit de saída é alterado com uma probabilidade de $1/2$ sempre que um simples bit de entrada é complementado [WEB 86].

BIC-Critério de Independência do Bit de Saída: Uma função que satisfaz o BIC tem a seguinte característica: sempre que um bit de entrada é complementado, os coeficientes da correlação entre cada dois bits alterados é zero [DAW 92].

As caixas-S projetadas, levando em consideração estes critérios, apresentam características criptográficas importantes para a segurança dos algoritmos, sendo consideradas boas Caixas-S [ADA 90].

Nas tabelas 3.4 e 3.5, encontram-se os critérios adotados no projeto de Caixas-S dos cifradores estudados.

Tabela 3.4: Critérios Adotados em Projetos de Caixas-S

Cifrador	Critérios Adotados
DES	Cada Caixa-S recebe como entrada 6 bits e fornece 4 bits de saída. Nenhum bit de saída das Caixas-S deveria ser uma função linear dos bits de entrada. Cada linha das Caixas-S deveria incluir todas as 16 possíveis combinações dos bits de saída. Se duas entradas de uma Caixa-S diferirem exatamente de um bit, as saídas precisam diferir em pelo menos dois bits. Se duas entradas de uma Caixa-S diferirem exatamente em dois bits intermediários, as saídas precisam diferir em pelo menos dois bits. Se duas entradas de uma Caixa-S diferirem nos seus dois primeiros bits e são idênticos em seus dois últimos bits, as duas saídas não necessitam ser a mesma. Para qualquer 6 bits não zero de diferentes entradas, não mais que 8 dos 32 pares de entradas devem expor estas diferenças na saída. Fonte: [COP 94]
CAST-256	Efeito Avalanche. SAC BIC Fonte: [ADA 99]

Tabela 3.5: Critérios Adotados em Projetos de Caixas-S - Continuação

Cifrador	Critérios Adotados
TWOFISH	Caixas-S dependentes da Chave. Fonte: [SCH 98]
MARS	A Caixa-S não deve conter palavras com conteúdo zero ou um. Dentro de cada duas Caixas-S, todas as entradas diferem em pelo menos três dos quatro bytes. S não contém duas entradas $S[i]$,$S[j]$ $(i \neq j)$, tal que $S[i] = S[j]$,$S[i] = \neg S[j]$ ou $S[i] - S[j]$. S possuí $(\frac{512}{2})$ distintas diferenças XOR e $2 \times (\frac{512}{2})$ distintas diferenças subtração. Para quaisquer duas entradas em S estas diferem em pelo menos 4 bits. Requer que a paridade parcial de S seja pelo menos $1/30$. Requer que um simples bit parcial de S seja pelo menos $1/30$. Requer que a dois bits parciais de S seja pelo menos $1/30$. Minimizar estas características para todas as Caixas-S que satisfazem os critérios anteriores. Fonte: [BUR 99]
SERPENT	Efeito Avalanche. Boa característica Linear, com probabilidade de pelo menos $1/4$. Boa característica Diferencial, com probabilidade entre $1/2 \pm 1/4$. Fonte: [AND 98]
RIJNDAEL	Inversível. Minimização da grande correlação não trivial entre a combinação linear dos bits de entrada e combinação linear dos bits de saída. Minimização dos grande valores não triviais da tabela EXOR (Ou-exclusivo). Complexidade de expressões algébricas em $GF(2^8)$. Fonte: [DAE 99]

3.6 Conclusão

Este capítulo apresentou o histórico do projeto de Caixas-S e sua importância nos projetos de cifradores de bloco simétricos, bem como as principais técnicas e critérios de desenvolvimento adotados pelos cifradores estudados no capítulo anterior. Pudemos verificar quais são os critérios que são considerados para classificar uma Caixa-S. Também podemos concluir que as técnicas mais utilizadas são: Manual, Randômica e Matemática.

Com base nos estudos das características de projeto das Caixas-S dos crifradores analisados, foi construida uma tabela comparativa entre si.

Capítulo 4

Não Linearidade de Caixas-S

4.1 Introdução

O objetivo deste capítulo é apresentar os principais conceitos e ferramentas matemáticas que são utilizadas para avaliar as Caixas-S através do critério de não linearidade.

Começamos fazendo uma breve introdução as Caixas-S e funções booleanas, em seguida apresentamos a característica de não linearidade de Caixas-S, funções booleanas e o cálculo da não linearidade através da Transformada Walsh-Hadmard. Por final são apresentados os resultados conhecidos de não linearidade de Caixas-S de 8×8.

4.2 Conceitos

Nos cifradores de bloco, cada rodada consiste de operações de transposição e substituição utilizando-se de um conjunto fixo de tabelas auxiliares, conhecidas como Caixas-S. A transposição é uma operação linear e a substituição é a única operação não linear dos cifradores.

As substituições S (Caixas-S) podem ser implementadas como um circuito e modeladas como um sistema de equações booleanas. Utilizando esta representação funcional é possível avaliar uma Caixa-S em função das propriedades apresentadas pelas

funções booleanas, propriedades estas que são muito importantes para a segurança dos sistemas criptográficos.

A segurança dos cifradores de bloco é analisada estudando as propriedades apresentadas pelas suas Caixas-S como um conjunto de funções booleanas, ou seja, nos permite avaliar o poder do sistema criptografico através de suas Caixas-S.

4.3 O que é Não Linearidade

A linearidade é considerada uma fraqueza significativa em sistemas criptográficos, a não linearidade é uma forma explícita de avaliar a falta de fraquezas no sistema criptográfico. A não linearidade de uma Caixa-S representa um indicador chave para avaliar o poder de um cifrador de blocos [DAW 91].

No caso dos sistemas criptográficos, é desejado que a não linearidade de uma tabela de substituição (Caixa-S) seja o valor mínimo para cada bit de saída da tabela [RIT 98b].

O critério de não linearidade permite avaliar o poder de uma Caixa-S com relação a criptoanálise linear e diferencial. A criptoanálise linear explora a baixa não linearidade de uma Caixa-S e a criptoanálise diferencial explora as entradas na tabela de distribuição de diferenças [JS 94b].

Para avaliar a não linearidade é necessário armazenar os resultados da função para cada possível combinação das variáveis de entrada. Para uma tabela de 4-bits é necessário armazenar e transformar 16 (2^4) elementos, para uma tabela de 8-bits é necessário armazenar e transformar 256 (2^8) elementos, para uma tabela de 16-bits são necessários transformar e armazenar 65536 (2^{16}) elementos. Desta forma, avaliar funções com grande quantidade de bits de forma rápida é uma tarefa impossível, sendo assim, não podemos avaliar a não linearidade de cifradores de blocos (64-bits, 128-bits, 256-bits, etc). Em contra partida podemos avaliar a não linearidade das tabelas de substituição (Caixas-S) que são compostas por um vetor de 2^n de m-bits [RIT 98b].

Se considerarmos uma Caixa-S como um conjunto ordenado de saídas de uma função booleana, a utilização de transformadas rápidas para calcular as pro-

priedades criptográficas desejadas nos é habilitada [WM 99, LB 00], mais especificamente a Transformada Rápida Walsh-Hadamard[1], sendo esta a ferramenta utilizada para calcular a não linearidade [RIT 98c].

4.4 Funções Booleanas

A seguinte definição é dada para funções booleanas [DAG 95]:

"Seja *B* uma Álgebra de Boole e, sejam $x_1, ..., x_n$ variáveis tais que seus valores pertencem a *B*. Chama-se *função booleana* de *n* variáveis a uma aplicação *f* de B^nem *B* satisfazendo as seguintes regras:

- Se para quaisquer valores de $x_1, ..., x_n$, $f(x_1, ..., x_n) = a, a \in B$, então *f é uma função booleana. É a função constante;*
- *Se para quaisquer valores de* $x_1, ..., x_n$, $f(x_1, ..., x_n) = x_i$ *para algum i com* $(i = 1, ..., n)$*, então f é uma função booleana. É a função projeção;*
- *Se f é uma função booleana, então g definida por* $g(x_1, ..., x_n) = (f(x_1, ..., x_n))'$ *para todo* $x_1, ..., x_n$ *é uma função booleana;*
- *Se f e g são funções booleanas, então h e k, definidas por* $h(x_1, ..., x_n) = f(x_1, ..., x_n) + g(x_1, ..., x_n)$ *e* $k(x_1, ..., x_n) = f(x_1, ..., x_n) \cdot g(x_1, ..., x_n)$ *para todo* $x_1, ..., x_n$ *são funções booleanas;*
- *Qualquer função construída por um número finito de aplicações das regras anteriores e somente tal função é booleana".*

Então, uma função booleana é qualquer função que pode ser construída a partir das funções constantes e projeção mediante um número finito das operações [' (NÃO), + (OU) e · (É)].

Uma função booleana que apresenta a seguinte forma: $f : 0, 1^n \rightarrow 0, 1^m$, é chamada de função booleana com *n* entradas e *m* saídas. Um função com *n* entradas é conhecida como função booleana com *n* variáveis.

[1]FWT - Fast Walsh-Hadamard Transform

Uma função booleana produz como resultado um único bit para cada possível combinação de valores de variáveis booleanas.

As funções booleanas são utilizadas nos sistemas modernos de criptografia como uma ferramenta para sua avaliação, pois estas apresentam propriedades importantes e desejadas aos sistemas. Uma das propriedades mais importantes apresentada pelas funções booleanas é a não linearidade.

4.5 Cálculo da Não Linearidade

Abaixo seguem as ferramentas matemáticas utilizadas para calcular a não linearidade.

4.5.1 Transformada Walsh-Hadamard

A transformada Walsh-Hadamard de uma função $f : \{0,1\}^n \rightarrow \{0,1\}$ é definida por: $W_f(w) = \sum_{x=0}^{2^n-1} (-1)^{f(x) + w \bullet x}$, onde $w \bullet x = w_{n-1}x_{n-1} \oplus ... \oplus w_0 x_0$, com $w, x \in \{0,1\}^n$, produto interno no espaço de dimensão n, $\mathbb{F}_2$.

A transformada Walsh-Hadamard é análoga à transformada de Fourier, sendo usada para vetores discretos (neste caso, binários) ao invés de contínuos [STA 99].

4.5.2 Distância de Hamming

Os códigos de Hamming são uma classe importante dos códigos utilizados na correção de erros que podem ser facilmente codificados e decodificados. Foram originalmente utilizados para controlar os erros em chamadas telefônicas de longa distância [TRA 01].

Definição [LIN 98]:

Se $x \in Q^n$, $y \in Q^n$, então a distância $d(x,y)$ de x e y é definida por $d(x,y) := |\{i | 1 \leq i \leq n, x_i \neq y_i\}|$, onde Q é um alfabeto com q distintos símbolos.

O peso $w(x)$ de x é definido por $w(x) := d(x,0)$.

A distância definida acima é conhecida como Distância de Hamming e o peso definido é conhecido como Peso de Hamming.

A distância de Hamming é uma das formas de avalir a correlação entre duas funções booleanas, comparando as suas tabelas verdade e contando o número de bits em que diferem.

O peso de Hamming de uma função booleana é o número de 1's existentes em sua tabela verdade.

4.5.3 Não Linearidade de Funções Booleanas

A não linearidade indica a distância de Hamming entre a função $f(x)$ e todas as funções lineares e seus complementos (também conhecidas como funções afin).

Definição [WM 97]:

Seja $f(x)$ um tabela verdade binária de uma função boolenada de n variáveis com sua transformada Walsh-Hadamard dada por $\hat{F}(w) = \sum_x (-1)^{f(x)} \oplus L_w(x)$, onde $L_w(x) = w_1x_1 \oplus w_2x_2 \oplus ... \oplus w_nx_n$ denota a função linear selecionada por w. O valor de $\hat{F}(w)$ é diretamente proporcional à correlação que $f(x)$ tem com $L_w(x)$: $c(f, L_w) = \frac{\hat{F}(w)}{2^n}$.

Seja WH_{max} o valor máximo absoluto de $\hat{F}(w)$, a não linearidade de $f(x)$ é dada por $N_f = \frac{1}{2}(2^n - WH_{max})$.

Ou seja, a transformada de Walsh-Hadamard avalia a correlação entre uma função booleana e o conjunto de todas as funções lineares e seus complementos, de forma rápida.

No trabalho de [MEI 90], é apresentado que a não lineariade máxima para n par, é dada por: $N_{max} = 2^{n-1} - 2^{\frac{n}{2}-1}$. Ou seja, para uma Caixa-S (4 $\times$ 4) a não linearidade máxima possível é $N_{max} = 6$, para $n = 6$ temos $N_{max} = 28$, para $n = 8$ temos $N_{max} = 120$. Para n impar $n \geq 9$ até o momento não foi encontrado um limite máximo. Para para $n = 3, 5, 7$ a não linearidade máxima é 2,12 e 56 respectivamente, a melhor solução apresentada para n impar é dada por: $N \leq 2\lfloor 2^{n-2} - 2^{\frac{n}{2}-2} \rfloor$ não sendo válida para $n = 7$ e $n = 15$ [WM 98].

Seja S^{-1} a Caixa-S inversa de uma Caixa-S. Foi demostrado em [NYB 92] que $NL(S^{-1}) = NL(S)$, ou seja, o valor de não linearidade de uma Caixa-S inversa é o mesmo de sua Caixa-S.

4.6 Resultados Conhecidos

Abaixo seguem os resultados alcançados por diversas técnicas de projeto de Caixas-S e sua avaliação através da não linearidade para Caixas-S de $8x8$.

Em [HEY 94], foram geradas 200 Caixas-S de forma randômica utilizando os critérios desejados e conseguiram alcançar a não linearidade de $94(38, 50\%)$, $96(23, 50\%)$ e $98(5, 50\%)$.

Em [RIT 98b], foram geradas $1.000.000$ Caixas-S de forma randômica e os seguinte resultados foram alcançados de não linearidade, $92(2, 50\%)$, $94(5, 00\%)$, $96(10\%)$, $98(25, 00\%)$, $100(35, 00\%)$, $102(20, 00\%)$, $104(2, 50\%)$.

Em [JS 94a], utilizando as técnicas desejadas, conseguiram alcançar a não lineraridade máxima de 116 dentre as Caixas-S projetadas.

Em [WM 97], foram geradas 1.000, 10.000, 100.000 e $1.000.000$ de Caixas-S usando AG sobre a função booleana da Caixa-S e conseguiram alcançar as seguintes não linearidades: 110 e 112, respectivamente. Neste mesmo trabalho, foram empregadas AG com Hill Climbing, e para uma população de 1.000 a 10.000 o resultado máximo obtido de não linearidade foi de 114.

Em [WM 98], utilizando técnicas baseadas em heurística (AG e Hill Climbing) a máxima não linearidade alcançada foi 116.

Na tabela 4.1 abaixo, temos o resumo dos resultados obtidos nos trabalhos apresentados anteriormente.

Pela tabela acima, podemos notar que os valores alcançados pelos projetos de Caixas-S e seus respectivos critérios estão se aproximando do limite superior teórico de 120, até o momento temos como limite superior prático o valor de não linearidade igual a 116.

Tabela 4.1: Principais Resultados Obtidos

Não Linearidade	Técnica	Trabalho	Ano
98	Randômica com Testes	[HEY 94]	1994
104	Randômica	[RIT 98b]	1998
112	AG	[WM 97]	1997
114	AG com Hill Climbing	[WM 97]	1997
116	Randômica com Testes	[JS 94a]	1994
116	Heurística (AG e Hill Climbing)	[WM 98]	1998

4.7 Conclusão

Este capítulo apresentou os conceitos de não linearidade, funções booleanas, a ferramenta utilizada para efetuar o cálculo de não linearidade de funções booleanas e como estas podem ser aplicadas as Caixas-S.

Com base no estudo foi elaborada uma tabela apresentando os principais resultados conhecidos de não linearidade para Caixas-S compatíveis com a do AES de 8 $\times$ 8.

Capítulo 5

Algoritmos Genéticos

5.1 Introdução

O objetivo deste capítulo é apresentar uma introdução aos Algoritmos Genéticos, com sua terminologia biológica e traçar seu paralelo com a teoria da evolução de Darwin. Em seguida, apresentar os principais elementos e operadores, bem como as suas aplicações em diversas áreas da ciência, em especial as aplicações de AG em Criptografia.

Este estudo dos Algoritmos Genéticos será utilizado como base para o projeto de Caixas-S proposto neste trabalho.

5.2 Introdução aos Algoritmos Genéticos

Segundo [BIT 98], "Inteligência Artificial (IA) é um ramo da ciência de computação ao mesmo tempo recente e muito antigo, pois, a IA foi construída a partir de idéias filosóficas, científicas e tecnológicas herdadas de outras ciências, algumas tão antigas quanto à lógica. Sendo, o objetivo central da IA simultaneamente teórico - a criação de teorias e modelos para a capacidade cognitiva - e prático - a implementação de sistemas computacionais baseados nestes modelos."

A IA tem muitas linhas de atuação e, neste trabalho, iremos abordar uma

de suas linhas: a computação evolutiva. Esta é uma área de pesquisa multidisciplinar que envolve a biologia, inteligência artificial, otimização numérica e engenharia [BÄC 96].

A Computação Evolutiva pode ser definida como: "desenvolvimento de computações artificiais inspiradas nas *computações* biológicas realizadas pelos seres vivos para viver e se reproduzir" [BIT 98].

Um histórico sobre o desenvolvimento da área de computação evolutiva pode ser encontrado no trabalho de Melanie Mitchell [MIT 97], no qual discute, também, exemplos de aplicação desta linha de pesquisa.

A Computação Evolutiva permite a simulação do processo de evolução natural. As idéias que permitem esta simulação são as seguintes [BÄC 96][BIT 98]:

- A criação de uma população que seria uma solução hipotética, possivelmente obtida na sua primeira geração de modo aleatório, na qual estes indivíduos apresentam, de modo intrínseco, os parâmetros que descrevem, não somente uma possível solução ao problema proposto no espaço de potenciais soluções.

- A criação de uma entidade, chamada *função de avaliação*, capaz de julgar a aptidão de cada um dos indivíduos no espaço de potenciais soluções. Essa entidade não precisa deter conhecimento sobre como encontrar uma solução para o problema, mas apenas atribuir uma *nota* ao desempenho de cada um dos indivíduos da população.

- Utilização de uma série de operadores que são aplicados à população de uma dada geração para obter os indivíduos da próxima geração. Estes operadores são baseados nos fenômenos que ocorrem na evolução natural. Os principais operadores são: *seleção*, *recombinação* e *mutação*.

Os algoritmos genéticos são um dos principais representantes do ramo da computação evolutiva, que tem como base a simulação dos processos vitais em computador, sendo este processo conhecido como *processos genéticos* [BÄC 96].

Os algoritmos genéticos foram inventados por John Holland e desenvolvidos por ele, seus alunos e companheiros na Universidade de Michigan, nas décadas

de 60 e 70. A pesquisa de Holland não estava centrada no desenvolvimento de um algoritmo para resolução de problemas, mas, sim, no estudo formal do fenômeno de adaptação que ocorre na natureza e como estes mecanismos de adaptação natural podem ser portados para computadores [MIT 97].

O campo de estudos foi batizado como algoritmos genéticos por Holland em referência à genética sua origem de estudos [DAV 91].

Os algoritmos genéticos tornaram-se populares através do trabalho de David E. Goldberg, um dos alunos de Holland, que utilizou esta técnica para resolver um problema complexo de controle de transmissão de gás em dutos [HAU 98].

Resumindo, os Algoritmos Genéticos são métodos de otimização e busca inspirados nos mecanismos da evolução de populações de seres vivos [LAC 99].

O campo dos AGs é representado por três grandes áreas de pesquisa:

- Algoritmos Genéticos Naturais;
- Otimização;
- Aprendizado de Máquinas, através de sistemas classificadores.

O mecanismo de funcionamento dos AG's torna este campo de pesquisa muito atrativo e aplicado em diversas áreas.

Segundo Goldberg [GOL 89], os AG's são diferentes dos mecanismos normais de otimização e busca nos seguintes pontos:

1. AG's trabalham com uma codificação de conjunto de parâmetros, não com parâmetros individualizados;
2. AG's buscam por uma população de pontos e não por um simples ponto;
3. AG's se utilizam de informações da função objetivo e não de informações secundárias ou outra informação auxiliar;
4. AG's utilizam regras de transição probabilísticas e não determinísticas.

Na relação abaixo, podemos notar que a utilização de AG's apresenta as seguintes vantagens [HAU 98]:

- Otimiza, utilizando parâmetros contínuos e discretos;
- Dispensa informações secundárias;
- Efetua busca simultaneamente em uma grande área do espaço de busca;
- Trabalha com um grande número de parâmetros;
- É muito adequado para trabalhar em computadores de arquitetura paralela;
- Otimiza parâmetros extremamente complexos, escapa dos mínimos locais;
- Fornece uma relação de ótimos parâmetros e não somente uma possível solução;
- Manipula parâmetros codificados;
- Utiliza dados gerados numericamente, experimentalmente e funções analíticas;

Com a sua forma de trabalhar de acordo com a evolução dos seres vivos e por apresentar diversas vantagens sobre outros métodos, os algoritmos genéticos têm sido uma boa alternativa para auxiliar na solução de uma grande variedade de problemas.

5.3 Terminologia Biológica

Os algoritmos genéticos seguem os princípios da seleção natural e sobrevivência do mais apto, apresentados pelo naturalista e fisiologista Charles Darwin em seu livro *"A Origem das Espécies"*.

Segundo Darwin, "Quanto melhor um indivíduo se adaptar ao seu meio ambiente, maior será sua chance de sobreviver e gerar descendentes" [LAC 99].

Na biologia, a teoria da evolução nos diz que o meio ambiente seleciona, em cada geração, os seres vivos mais aptos de uma população para sobrevivência, em função disto, somente os mais aptos (fortes) conseguem se reproduzir, isto é, levando em

consideração a premissa de que os menos aptos são eliminados antes de gerarem novos descendentes.

Durante o processo de reprodução, ocorrem fenômenos que atuam sobre o material genético armazenado nos cromossomos, sendo estes fenômenos a mutação e a recombinação.

Estes fenômenos são responsáveis pelas mudanças ocorridas na população. Sobre esta população diversificada, age a seleção natural, permitindo a sobrevivência apenas dos seres mais aptos.

Um Algoritmo Genético é a metáfora desses fenômenos e, em função disto, os AG's possuem muitos termos oriundos da biologia.

Os principais termos biológicos utilizados pelos AG's são os seguintes [MIT 97] [LAC 99]:

Genoma: É o conjunto completo do material genético de um organismo vivo. Nos AG's, representa a estrutura de dados que codifica uma solução para o problema;

Cromossomo: É o elemento portador de material genético. Nos AG's, representa um simples ponto no espaço de busca ou um candidato à solução do problema;

Gene: Um cromossomo é composto de genes. O gene é o elemento que transmite a hereditariedade e controla as características do indivíduo. Nos AG's, o gene é um parâmetro codificado no cromossomo;

Posição (Locus): Posição em que um gene se encontra no cromossomo;

Alelo: Diferentes possibilidades de uma característica de um gene (Ex. Olhos azuis, verdes, castanhos , etc). Nos AG's, representa o valor que um gene pode assumir.

Indivíduo: Um simples membro da população. Nos AG's, um indivíduo é formado pelo cromossomo e sua aptidão;

Genótipo: Representa a composição genética contida no genoma. Nos AG's, representa a informação contida no cromossomo ou genoma;

Fenótipo: É originado pela interação (genótipo + meio). Nos AG's representa o objeto, estrutura ou organismo construído a partir das informações do genótipo;

Epistasia: Interação entre genes do cromossomo, isto é, quando um valor de gene influencia o valor de outro gene. Problemas com alta epistasia são de difícil solução em AG's.

Em função do seu mecanismo os AG's apresentam com jargão a mesma utilizada pela biologia para o processo da evolução das espécies.

5.4 Processo de Evolução

O processo de evolução que ocorre na natureza é efetuado através de um processo de evolução simulado, que é realizado através de um algoritmo matemático implementado em um computador [MOS 95].

As características da evolução que devemos levar em consideração nos projetos são as seguintes [DAV 91]:

- A evolução é um processo que opera nos cromossomos, no lugar dos seres vivos codificados por eles;
- A seleção natural é a relação entre cromossomos e suas estruturas codificadas. No processo de seleção natural, os cromossomos que codificam melhores estruturas se reproduzem com maior freqüência dos que não têm esta característica;
- O processo de reprodução é o ponto em que a evolução ocorre. A mutação pode causar diferenças entre os cromossomos dos filhos e dos pais e o processo de recombinação pode criar cromossomos bem diferentes nos filhos devido à combinação de material genético do cromossomo dos pais;
- O processo de evolução biológica não possui memória, pois não possui informações que lhe permitam voltar a etapas anteriores do processo.

Como ocorre na evolução biológica, a evolução baseada em simulação é um processo que será projetado para encontrar cada vez melhores cromossomos, através de uma manipulação aleatória de seu conteúdo.

O termo aleatório se refere ao fato do processo não ter nenhuma informação sobre o problema que está tentando resolver, exceto o valor da função objetivo.

A função objetivo ou função de aptidão é a única informação que temos sobre o cromossomo e esta informação é a sua nota, que reflete a qualidade da solução que ele representa.

Um AG pode ser visto como uma estrutura de controle que organiza e dirige um conjunto de transformações e operações indicadas para simular o processo de evolução [MOS 95].

5.5 Elementos dos Algoritmos Genéticos

Um algoritmo genético é constituído dos seguinte itens:

1. População de Cromossomos;
2. Seleção;
3. Recombinação;
4. Mutação.

Como podemos observar, um algoritmo genético é composto por estruturas bem simples e bem definidas.

5.5.1 População de Cromossomos

A população é formada por um conjunto aleatório de cromossomos, não é uma regra. Temos que considerar dentro da população também o seu tamanho, pois ele afeta o desempenho global e a eficiência dos AG's. Uma população muito pequena oferece uma pequena cobertura do espaço de busca, causando uma queda no desempenho.

Uma população suficientemente grande fornece uma melhor cobertura do domínio do problema e previne a convergência prematura para soluções locais. Entretanto, com uma grande população tornam-se necessários recursos computacionais maiores, ou um tempo maior de processamento do problema.

Um cromossomo é uma estrutura de dados, geralmente vetor ou cadeia de bits, que representa uma possível solução do problema. A forma de codificação baseada em cadeia de bits (0 e 1) é a mais tradicional [GOL 89], porém nem sempre é a melhor escolha [DAS 97].

De uma forma geral, um cromossomo representa um conjunto de parâmetros da função objetivo, cuja resposta será maximizada ou minimizada.

O conjunto de todas as configurações que o cromossomo pode assumir forma o seu *espaço de busca*. Se o cromossomo representa n parâmetros de uma função, então o seu espaço de busca é um espaço com n dimensões [LAC 99].

Um cromossomo computacional típico poderia ser $S_1 = 100101111010110$, onde por exemplo, cada 3 bits descrevem o status de um sensor de um robô que tem níveis de 0 a 7.

5.5.2 Seleção

O processo de seleção tem como objetivo escolher cromossomos em uma população para a reprodução. A fase de seleção descarta os cromossomos com baixa aptidão, desta forma, muitos genes mutados durante a evolução são eliminados.

Esta escolha pode ocorrer das seguintes formas tradicionais:

Aptidão: São escolhidos os elementos que apresentarem alta aptidão na população inicial, esta aptidão é calculada através da função de aptidão, que é um esquema de avaliação do fenótipo. Isto é, este mecanismo é inspirado na seleção natural que seleciona os mais aptos. Os mais aptos são aqueles que representam uma boa solução para o problema proposto. *Função de Aptidão* ou *Função Objetivo*, como são conhecidas no jargão dos AG's, são responsáveis por efetuar esta avaliação do cromossomo, podendo esta avaliação ser bastante complicada e demandando um alto custo

computacional. O processo de geração desta função de avaliação deve ser muito criterioso [HAU 98, LAC 99].

Roleta: Este método possui este nome por sua semelhança com a roleta utilizada em cassinos. A seleção pelo método da roleta é calculada através da soma total de todas as aptidões dos indivíduos de uma população. Para cada indivíduo divide-se sua aptidão pelo valor da soma total das aptidões de todos os indivíduos, chegando a probabilidade que cada indivíduo tem de ser selecionado dentro da população. Com as probabilidades montadas de cada indivíduo, podemos elaborar a roleta simplificada da figura 5.1, onde os indivíduos mais aptos ganham uma área proporcional a sua aptidão dentro da roleta. Fazendo a roleta girar, aqueles indivíduos mais aptos terão maiores chances de serem sorteados, pois apresentam uma área maior da roleta.

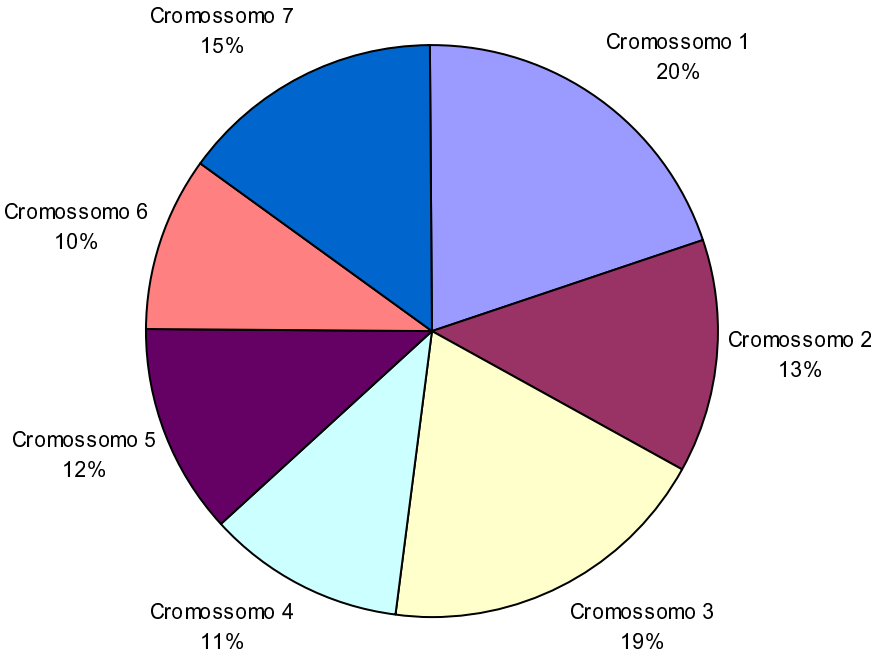

Figura 5.1: Seleção pelo Método da Roleta: Os cromossomos são classificados em função de sua aptidão, os indivíduos mais aptos ocupam uma área maior, aumentando a sua probabilidade de serem selecionados.

5.5.3 Recombinação (“Crossover”)

A recombinação é um dos principais mecanismos de busca dos AG’s para explorar regiões desconhecidas do espaço de busca [LAC 99].

Este operador é simples e visa realizar a troca de sub-partes entre dois cromossomos. Neste processo um par de cromossomos pai gera dois cromossomos filhos. Cada um dos cromossomos pais tem sua cadeia cortada em uma posição, conforme figura 5.2 ou em duas posições, conforme figura 5.3.

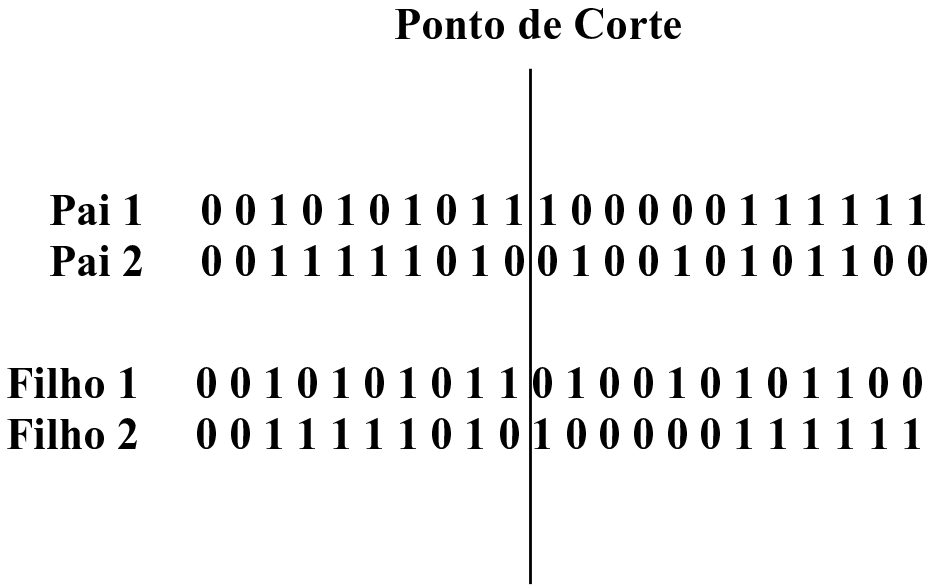

Figura 5.2: Recombinação em 1 Ponto: A parte de material genético que será recombinada entre os pais é estipulada pelo ponto de corte, que neste caso é realizado em somente um ponto. Os filhos resultantes apresentam em seu material genético parte do material do Pai1 e Pai2.

A recombinação é aplicada com uma dada probabilidade a cada par de cromossomos selecionados. Esta probabilidade utilizada fica entre 60% e 90% [LAC 99].

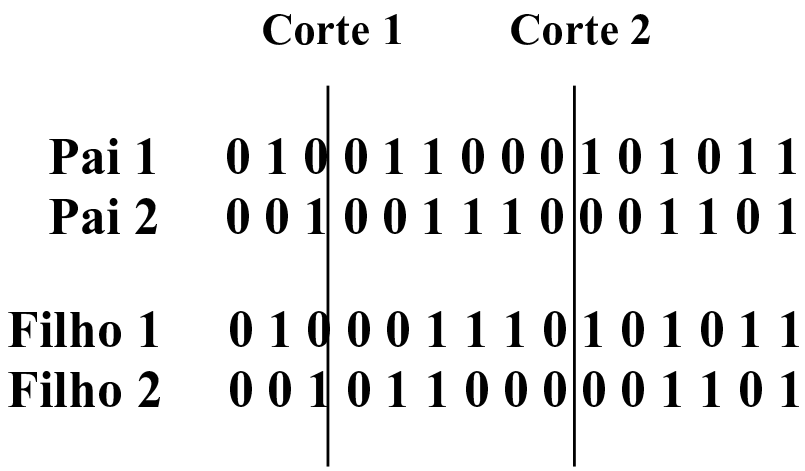

Figura 5.3: Recombinação em 2 Pontos: A parte de material genético que será recombinada entre os pais é estipulada pelo ponto de corte, que neste caso é realizado em dois pontos. Os filhos resultantes apresentam em seu material genético parte do material do Pai1 e Pai2.

5.5.4 Mutação

A mutação é outro mecanismo principal de busca dos AG's para explorar regiões desconhecidas do espaço de busca [LAC 99].

Este processo consiste na troca aleatória dos valores da *string* de um cromossomo. Assim como a mutação genética, a probabilidade de ocorrer é muito pequena e ocorre uma modificação no código genético. Nos AG's, a mutação pode ocorrer em qualquer posição na cadeia de *strings*. Esta operação pode ser visualizada, na ilustração da figura 5.4.

A mutação melhora a diversidade dos cromossomos da população inicial, em contra partida, destrói informações contidas no cromossomo. Deve-se aplicar uma taxa de mutação pequena entre $0,1\%$ a 5%, suficiente para assegurar a diversidade [LAC 99].

Em resumo, a mutação reflete uma taxa de alteração de um gene 0 em 1 ou vice-versa.

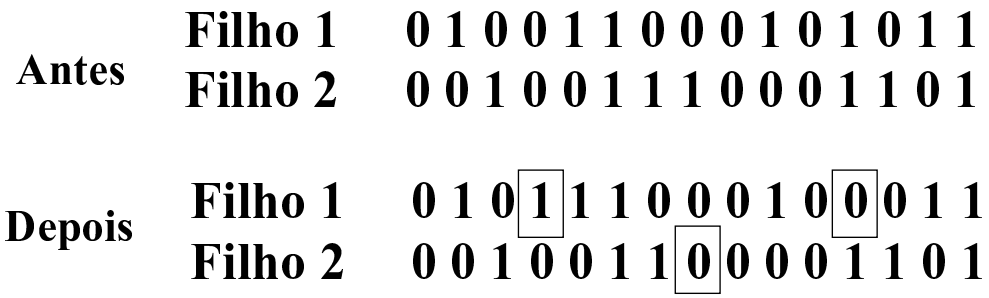

Figura 5.4: Mutação: Um elemento ou mais do cromossomo sofre alteração do seu valor, resultando em outro elemento.

5.6 Algoritmos Genéticos Simples

De uma forma bem simples, um algoritmo genético é constituído dos seguinte passos, aplicados interativamente [DAV 91, MIT 97]:

1. Inicie uma população aleatoriamente ou com indivíduos oriundos de outros processos;

2. Calcule a função de aptidão para cada cromossomo da população;

3. Selecione, da população, pares de cromossomos onde os cromossomos mais aptos têm maiores chances de serem escolhidos, cada cromossomo pode ser escolhido mais de uma vez. Os mais aptos apresentam maior valor em sua aptidão;

4. O par escolhido é recombinado com uma probabilidade P_r (*probabilidade de recombinação ou cruzamento*). O ponto de recombinação ao longo do cromossomo é escolhido aleatoriamente;

5. Cada *locus* do cromossomo está sujeito à mutação com uma probabilidade P_m (*probabilidade de mutação*);

6. Atualize a população com os resultados obtidos;

7. Se o critério de parada não for satisfeito, volte para o passo dois (2). O critério de parada é uma condição a ser adotada, por exemplo, quando 95% da população apresentar pouca variação em seus valores de aptidão.

Cada iteração deste processo é chamado de geração.

Estes passos podem ser escritos na forma de pseudocódigo [LAC 99].

"Seja $S(t)$ a população de cromossomos na geração t.

$t <-- 0$
inicializar $S(t)$
avaliar $S(t)$
enquanto o critério de parada não for satisfeito *faça*
{
$t <-- t + 1$
selecionar $S(t)$ a partir de $S(t-1)$
aplicar recombinação sobre $S(t)$
aplicar mutação sobre $S(t)$
avaliar $S(t)$
}
fim enquanto"

5.7 Aplicações de Algoritmos Genéticos

Os algoritmos genéticos têm sido muito aplicados em um grande número de aplicações científicas e de engenharia, podemos citar as seguintes áreas de aplicação [MIT 97]:

- Otimização;
- Programação Automática;
- Aprendizado de Máquinas;

- Economia;
- Sistemas Imunológicos;
- Ecologia;
- Genética;
- Evolução;
- Aprendizado;
- Sistemas Sociais.

O campo de atuação dos algoritmos genéticos tem sido muito vasto e, em pesquisas mais recentes, tem sido utilizado em conjunto com outras técnicas, como Lógica Fuzzy, Sistemas de Regras, Redes Neurais, Árvores de Decisão, Raciocínio Baseado em Casos, Data Mining, etc [LAC 99].

Maiores informações sobre aplicações envolvendo os algoritmos genéticos podem ser conseguidos em [GOL 89, DAV 91, DAS 97, HAU 98, BAN 99].

5.8 Algoritmos Genéticos e Criptografia

Esta característica de multidisciplinaridade dos AGs, fez com que trabalhos envolvendo AGs e Criptografia surgissem.

O AGs tem sido usado com sucesso, como uma ferramenta de criptoanálise [SPI 93]. Os cifradores clássicos (substituição e transposição) são tipicamente vulneráveis aos ataques dos AGs [MAT 93, SPI 93].

Nos trabalhos dos pesquisadores do Information Security Research Center da Universidade de Queensland, Austrália, os AGs tem sido utilizados para projetar Caixas-S com algo grau de difusão [WM 97, WM 98, WM 99].

No trabalho de Burnett [LB 00], são utilizados as técnicas de Hill Climbing (subida de encosta [LAC 99]) e AGs para projetar Caixas-S compatíveis com o cifrador

MARS, e provaram que as Caixas-S geradas, são superiores as Caixas-S geradas pelos projetistas do MARS.

5.9 Conclusão

Este capítulo apresentou um dos principais elementos da Computação Evolutiva, os Algoritmos Genéticos, com seus operadores e elementos baseados na teoria da evolução de Darwin. Também podemos observar como este paralelismo simples pode ser implementado computacionalmente e colher os frutos do seu mecanismo, obtendo bons resultados para a solução de uma gama de problemas em diversas áreas de pesquisa.

Com base neste capítulo foram estudadas as aplicações dos Algoritmos Genéticos na área de Criptografia e seus resultados.

Capítulo 6

Projeto de Caixas-S utilizando Algoritmos Genéticos

6.1 Introdução

O objetivo deste capítulo é apresentar quais foram as estratégias e critérios de implementação utilizados para a geração de Caixas-S utilizando-se da técnica de Algoritmos Genéticos, bem como os resultados alcançados.

6.2 Estratégias de Implementação

Para implementação de um AG para buscar uma solução desejada é necessário que os seus critérios e estratégias de implementação sejam definidos. Os critérios e métodos adotados encontram-se especificados nas subseções abaixo.

6.2.1 Critério de Representação da Solução

O primeiro passo a ser tomado durante a implementação de um processo evolucionário é decidir como a solução será representada. A forma clássica ou canônica será adotada neste trabalho, ou seja, a forma binária. Isto quer dizer que os candidatos a melhor solução para o problema serão representados na forma binária. A escolha da

forma binária também se deve ao cálculo da não linearidade, que utiliza os elementos na forma binária.

Vimos que para uma tabela de 8-bits é necessário armazenar e transformar 256 (2^8) elementos, o nosso candidato à solução será a representação dos elementos desta tabela na forma binária.

Nesta forma de representação temos um cromossomo de 2048 bits, onde a cada 8 bits temos um elemento codificado da tabela de substituições, conforme apresentado no exemplo da tabela 6.1.

Tabela 6.1: Exemplo de Representação

Elementos em Decimal	0, 1, 2, 3, 4,...,255
Elementos em Binário	00000000000000010000001000000011000000100,...,11111111

6.2.2 Critério de População Inicial

A população inicial utilizada no trabalho é gerada de forma aleatória, sendo também um dos parâmetros utilizados pela aplicação.

Neste trabalho, utilizamos como parâmetro inicial de população o tamanho de 20, ou seja, a população inicial será composta de 20 cromossomos de 2048 bits, com não linearidade igual ou superior a 100.

6.2.3 Critério de Seleção

A seleção dos cromossomos será dada pela função de sua aptidão, cada cromossomo será avaliado e receberá uma nota correspondente a sua aptidão como candidato à solução do problema. Neste trabalho, a função de aptidão utilizada será a própria não linearidade: os elementos que tiverem não linearidade menor a 100 serão desconsiderados, por não representarem uma boa solução com relação ao critério de avaliação adotado, porém contabilizados para efeitos de estatísticas.

A representação binária habilita a utilização da transformada rápida de Walsh-Hadamard para efetuar o cálculo da não linearidade, as implementações em Pascal

e Java do cálculo da não linearidade, na qual este trabalho foi baseado, se encontram em [RIT 98c] e [RIT 98a], respectivamente.

6.2.4 Critério de Recombinação ou Crossover

A recombinação utilizada neste trabalho consiste em combinar (C_2^n) todos os elementos da população entre si, aos pares, seguindo os critérios de recombinação implementados, sendo este critério mais um dos parâmetros da aplicação.

Critérios de Recombinação utilizados neste trabalho:

- Aleatório: O ponto de corte para realizar a troca de subpartes entre os cromossomos é escolhido aleatoriamente. Esta é a forma clássica de implementação.

- Corte fixo em 1 ponto (25%): O ponto de corte para realizar a troca de sub-partes entre os cromossomos é realizado em 25% do cromossomo. Esta estratégia foi adotada com o objetivo de preservar material genético que pode ser perdido pelo corte aleatório e também verificar se desta forma encontramos cromossomos melhores.

- Corte fixo em 1 ponto(50%): O ponto de corte para realizar a troca de sub-partes entre os cromossomos é realizado em 50% do cromossomo. Esta estratégia foi adotada com o objetivo de preservar material genético que pode ser perdido pelo corte aleatório e também verificar se desta forma encontramos cromossomos melhores.

- Corte fixo em 1 ponto(75%): O ponto de corte para realizar a troca de sub-partes entre os cromossomos é realizado em 75% do cromossomo. Esta estratégia foi adotada com o objetivo de preservar material genético que pode ser perdido pelo corte aleatório e também verificar se desta forma encontramos cromossomos melhores.

- Corte Fixo em 2 pontos: O ponto de corte para realizar a troca de sub-partes entre os cromossomos é realizada, informando a posição do primeiro corte e segundo corte, será utilizado para a troca de conteúdo genético a parte entre os pontos de corte. Esta estratégia foi adotada com o objetivo de verificar se existem partes do

cromossomo que afetam a não linearidade do cromossomo mais do que outras e também verificar se desta forma encontramos cromossomos melhores.

- Corte Aleatório em 2 pontos: O ponto de corte para realizar a troca de sub-partes entre os cromossomos é realizada, escolhendo de forma aleatória a posição do primeiro e segundo corte, somente sendo especificado a quantidade de bits que se encontram entre o primeiro e segundo corte, será utilizado para a troca de conteúdo genético a parte entres os pontos de corte. Esta estratégia foi adotada com o objetivo de verificar se existem partes do cromossomo que afetam a não linearidade do cromossomo mais do que outras e também verificar se desta forma encontramos cromossomos melhores.

6.2.5 Critério de Mutação

A mutação não foi implementada como um processo formal no algoritmo utilizado, mas acaba ocorrendo com uma taxa muito elevada. Após a recombinação, o filho resultante apresenta parte do cromossomo A e parte do cromossomo B. Este novo cromossomo deve apresentar em sua formação os 2^8 elementos codificados possíveis, sem que ocorra repetição de nenhum destes elementos. Quando ocorre repetição, seja de um ou mais elementos, estes elementos são substituídos (mutados) para elementos que não constam no novo cromossomo. Esta mutação forçada ocorre de forma aleatória até não termos mais elementos repetidos no cromossomo.

Esta necessidade de não termos elementos repetidos em nosso cromossomo se deve à característica das Caixas-S de não apresentarem elementos repetidos em sua formação.

Esta característica fez com que a mutação seja implementada de forma obrigatória em cada cromossomo que é gerado.

6.3 Implementação

6.3.1 Parâmetros Utilizados

Abaixo seguem os parâmetros necessários para que o processo seja iniciado.

O algoritmo trabalha com os seguintes parâmetros:

- Número de Gerações [*Ng*]: Para quantas gerações o processo irá ocorrer.
- Critério de Recombinação [*Cr*]: Informa como será o processo de recombinação.
- População Inicial [*Pi*]: Informa o nome do arquivo que contém a população inicial, para o caso de possuirmos uma população inicial resultante de outro processo ou sugerida.
- População Final [*Pf*]: Informa o nome do arquivo que contém os elementos selecionados pela sua função de aptidão, durante as gerações requeridas no algoritmo.

6.3.2 Algoritmo Implementado

A solução implementada pode ser representada pelos seguintes passos:

1. Receber os parâmetros da aplicação [*Ng*, *Cr*, *Pi*, *Pf*].
2. Gerar a população inicial com 20 elementos. Caso *Pi* contenha algum elemento, será gerado somente a diferença, até atingirmos 20 elementos.
3. Efetuar a recombinação entre os elementos da *Pi* combinando-os entre si, conforme o critério dado por *Cr*.
4. Para cada elemento resultante da recombinação, realizar a mutação dos elementos codificados que se encontram repetidos no cromossomo, removendo a repetição de elementos.

5. Cada elemento gerado pela recombinação é avaliado através da função de aptidão (não linearidade). Se for um candidato considerado apto será armazenado como integrante da próxima geração, ou seja, somente os mais aptos são armazenados (não linearidade superior ou igual a 100).

6. Se o critério de parada *Ng* for satisfeito, finalizar a aplicação, caso contrário, os elementos aptos passam a ser a nova população inicial e o processo volta ao passo três (3).

Ao final deste processo, teremos os elementos que foram evoluindo através de suas gerações e conseguiram atender aos critérios solicitados até a ultima geração solicitada.

6.3.3 Linguagem Utilizada

Para realizar a implementação de nosso algoritmo foi utilizada a linguagem de programação C++. A escolha da linguagem se deu em função de sua portabilidade e performance, bem como para poder gerar uma implementação utilizando o paradigma de orientação a objetos. Os programas fontes desta implementação orientada a objetos se encontram no Apêndice A. A implementação escrita em linguagem ”C”por [SMI 94] serviu como base inicial para este trabalho.

O compilador utilizado para a implementação foi o Microsoft Visual C++, esta decisão se deu em função do ambiente de testes disponíveis.

Abaixo, na Figura 6.1, temos o diagrama UML (Uniform Modeling Language) com as classes de nossa implementação.

6.3.4 Problemas Enfrentados

Ao testar a primeira versão da aplicação, que utilizava as estruturas em memória, nos deparamos com os seguintes problemas:

- A aplicação consumia 100% da CPU e 100% da memória, gerando problemas para os sistemas operacionais utilizados nos testes Windows 95, Windows 98, Windows

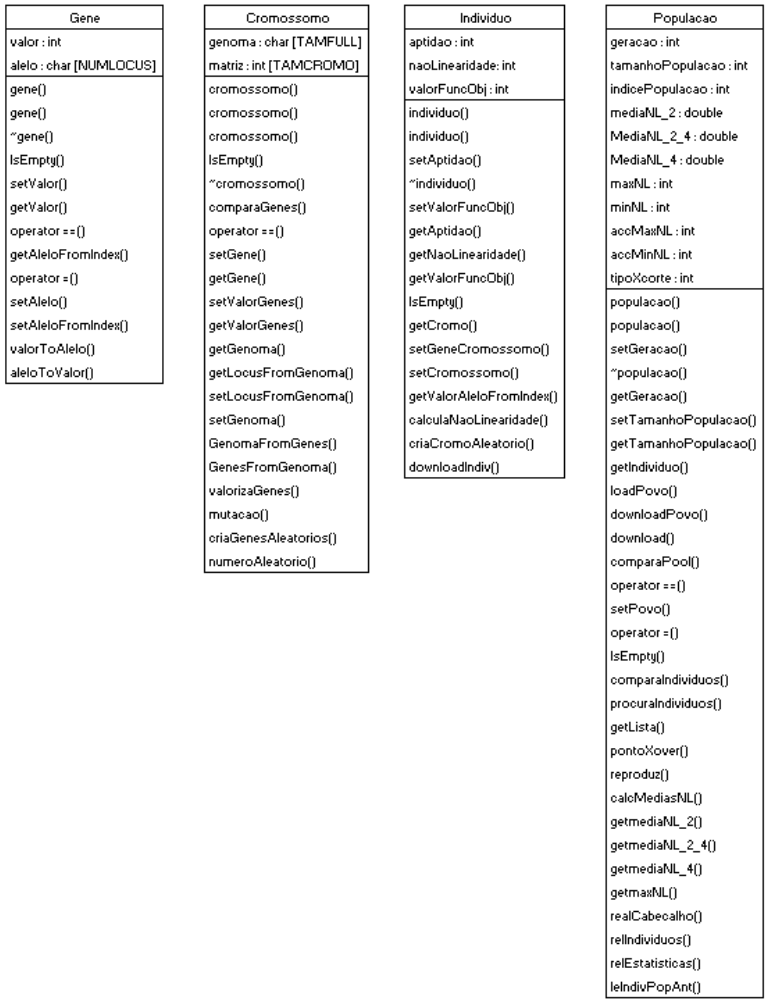

Figura 6.1: Diagrama UML da Implementação: Representação da implementação utilizando a notação UML dos objetos envolvidos na solução.

NT Server, Windows NT Workstation, Windows 2000 Professional e Windows 2000 Server.

- Não conseguiamos testar para mais de duas gerações, pois todos os recursos do sistema operacional eram consumidos, interrompendo a aplicação.

- Gerava problemas de paginação nos sistemas operacionais descritos anteriormente, interrompendo a aplicação.

Para contornar os problemas apresentados, resolvemos armazenar as estruturas no disco rígido em arquivos ao invés de memória, esta mudança deixou a aplicação mais estável, porém mais lenta.

Para testar a aplicação foram utilizados os seguintes ambientes:

- 20 máquinas Pentium 166 com 32 Mb de memória RAM e sistema operacional Windows 98.
- 8 máquinas Pentium III 400 Mhz com 128 Mb de memória RAM e sistema operacional Windows 2000 Server.

Utilizando os quatro critérios possíveis de recombinação e com evolução até a quinta geração, realizando somente 10 testes de cada tipo, obtivemos os seguinte tempos médios de processamento:

- Pentium 166: 30 horas.
- Pentium III: 18 horas.

Como pretendemos fazer no mínimo 1000 testes para cada critério possível de recombinação esta performance se torna um problema muito sério.

Foi feita mais uma implementação, incluindo os critérios de recombinação utilizando dois pontos de corte (fixo e aleatório), bem como uma revisão no algoritmo, decidiu-se eliminar os cromossomos que não eram utilizados (não aptos), antes eram armazenados, isto consumia bastante recurso de máquina, pois o arquivo crescia muito (chegando a 12 Mb), esta alteração fez com que problemas com gerenciamento de disco e memória do Windows 2000 Server diminuíssem e melhorassem a performance do algoritmo.

Diversas revisões foram feitas no código para melhorar o seu desempenho, sendo os resultados apresentados, frutos da ultima versão da aplicação com as implementações realizadas.

Como alternativa para resolver o problema de processamento e disponibilidade de hardware para os testes, o NPD da UFSC foi procurado com o objetivo de

utilizar o Super Computador utilizado para processamento científico, mas, infelizmente o mesmo estava em manutenção e até a finalização deste trabalho esta situação não havia sofrido alteração.

6.3.5 Duração dos Testes Realizados

Os testes com os critérios de recombinação implementados foram realizados em etapas, devido à disponibilidade de máquina para poder realizar os testes.

Estes testes foram realizados durante os períodos em que o laboratório da Fundação Softville não estava sendo ocupado.

Etapas dos Testes:

Etapa 1 : Início dia 27/03/2002 às 19h09min e término dia 12/04/2002 às 08h00min, sendo utilizadas 8 máquinas Pentium III 400 Mhz com 128 Mb de memória RAM e sistema operacional Windows 2000 Server.

Etapa 2 : Início dia 15/04/2002 às 16h35min e término dia 18/04/2002 às 17h00min, sendo utilizadas 8 máquinas Pentium III 400 Mhz com 128 Mb de memória RAM e sistema operacional Windows 2000 Server.

Etapa 3 : Início dia 27/04/2002 às 18h35min e término dia 03/05/2002 às 11h45min, sendo utilizadas 6 máquinas Pentium III 400 Mhz com 128 Mb de memória RAM e sistema operacional Windows 2000 Server.

Etapa 4 : Início dia 11/05/2002 às 12h40min e termino dia 19/05/2002 às 17h00min, sendo utilizadas 5 máquinas Pentium III 400 Mhz com 128 Mb de memória RAM e sistema operacional Windows 2000 Server.

Todos estas etapas totalizaram 769 horas e 16 minutos, aproximadamente 33 dias contínuos de geração e avaliação de Caixas-S.

6.3.6 Ferramenta

Para poder auxiliar na apuração dos resultados, foi feita a implementação de uma ferramenta que efetua a contabilização dos resultados alcançados.

A ferramenta trabalha sobre os arquivos contendo os resultados alcançados para cada tipo de teste, ou seja, ele contabiliza os totais por tipo de parâmetro, estes dados compilados são utilizados para formar as tabelas com os resultados alcançados.

Se esta ferramenta não fosse criada, o trabalho teria que ser realizado de forma manual, ou seja, abrir arquivo por arquivo e verificar as quantidades alcançadas, isto geraria problemas com a precisão dos testes e demandaria um tempo razoável para realizar a tarefa.

A ferramenta foi implementada também utilizando o Microsoft Visual C++.

6.4 Resultados Alcançados

Abaixo, seguem os resultados alcançados até a quarta etapa dos testes, até este momento não foi possível realizar todos os testes desejados, que são 100 rodadas para cada tipo de parâmetro.

Os testes foram divididos em três grupos, em função da característica dos parâmetros.

6.4.1 Corte Aleatório em 1 Ponto

Na Tabela 6.2 abaixo, temos os resultados alcançados para os testes com 1 ponto de corte escolhido em função do parâmetro especificado, com recombinação de 1 geração a 5 gerações.

Os parâmetros possíveis para os testes deste grupo são o tipo de corte, que pode assumir os seguintes valores:

- 0 - Ponto de Corte Aleatório no cromossomo;

- 1 - Ponto de Corte é realizado em 1/2 do cromossomo;
- 2 - Ponto de Corte é realizado em 1/4 do cromossomo;
- 3 - Ponto de Corte é realizado em 3/4 do cromossomo.

Tabela 6.2: Resultados do Corte Aleatório em 1 Ponto de 1 Geração e 4 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
1	0	100	2304	1239	133	0	30	104	40000
1	1	100	3163	1583	169	0	32	104	139900
1	2	100	2808	1382	195	0	32	104	138901
1	3	100	3747	1861	217	7	28	106	139900
Total		400	12022	6065	714	7	28	106	458701

Analisando os resultados alcançados na Tabela 6.2, podemos verificar para 1 Geração que:

- O corte do tipo 0 em suas 100 rodadas apresentou não linearidade máxima de 104, não linearidade mínima de 30 para uma população de 40.000 elementos analisados. O desempenho deste tipo de corte foi inferior aos demais por produzir uma população pequena.
- Os cortes do tipo 1,2 e 3 apresentaram uma população total final bem similar.
- O corte do tipo 3 apresentou elementos com menor não linearidade, igual a 28.
- O corte do tipo 3 apresentou, também, 7 elementos com a maior não linearidade alcançada, igual a 106.
- Este resultado já é melhor do que o alcançado no trabalho de Terry Ritter [RIT 98b].

Analisando os resultados alcançados na Tabela 6.3, podemos verificar para 2 Gerações que:

Tabela 6.3: Resultados do Corte Aleatório em 1 Ponto de 2 Gerações e 4 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
2	0	100	6473	3048	310	2	32	106	151684
2	1	100	4789	2162	243	0	32	104	325080
2	2	100	3598	1548	155	0	32	104	277218
2	3	100	7402	3216	324	2	32	106	384255
Total		400	22262	9974	1032	4	32	106	1138237

- O corte do tipo 0 em suas 100 rodadas apresentou não linearidade máxima de 106, não linearidade mínima de 32 para uma população de 151.684 elementos analisados. O desempenho deste tipo de corte foi superior aos demais por produzir em uma população pequena, em relação as demais, elementos com boa não linearidade.
- Os cortes do tipo 1e 3 apresentaram uma população total final bem similar.
- O corte do tipo 2 apresentou uma população final menor que a 1 e 3 e alcançou a mesma não linearidade que ambas.
- O corte do tipo 3 apresentou também 2 elementos com a maior não linearidade alcançada, igual a 106.
- Notamos que apesar de termos aumentado a quantidade de gerações, a não linearidade máxima alcançada continua sendo 106.

Tabela 6.4: Resultados do Corte Aleatório em 1 Ponto de 3 Gerações e 4 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
3	0	100	21646	9419	820	5	32	106	234788
3	1	100	6018	2488	262	5	32	106	462855
3	2	100	3894	1687	204	0	32	104	341200
3	3	100	13876	5729	580	4	32	106	667791
Total		400	45434	19323	1866	14	32	106	1706634

Analisando os resultados alcançados na Tabela 6.4, podemos verificar para 3 Gerações que:

- Os cortes do tipo 0, 1 e 3 em suas 100 rodadas apresentaram não linearidade máxima de 106 e não linearidade mínima de 32.
- O corte do tipo 2 apresentou não linearidade máxima de 104.
- O corte do tipo 0 apresentou uma população final menor que a 1 e 3 e alcançou a mesma não lineridade que ambas.
- O corte do tipo 3 apresentou elementos com a maior não linearidade alcançada, igual a 106 para uma populacão de 667.791, bem superior aos cortes 0, 1 e 3.
- Notamos que apesar de termos aumentado a quantidade de gerações a não linearidade máxima continua sendo 106, apesar do aumento da quantidade destes elementos.

Tabela 6.5: Resultados do Corte Aleatório em 1 Ponto de 4 Gerações e 4 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
4	0	100	23636	9968	944	10	32	106	161783
4	1	100	6154	2494	281	3	32	106	506470
4	2	100	3648	1637	176	0	32	104	339141
4	3	100	26852	10703	1028	10	30	106	845712
Total		400	60290	24802	2429	23	30	106	1853106

Analisando os resultados alcançados na Tabela 6.5, podemos verificar para 4 Gerações que:

- Os cortes do tipo 0, 1 e 3 em suas 100 rodadas apresentaram não linearidade máxima de 106.
- O corte do tipo 2 apresentou não linearidade máxima de 104.
- O corte do tipo 0 apresentou uma população final menor que a 1 e 3 e alcançou a mesma não linearidade que ambas.
- O corte do tipo 3 apresentou elementos com a maior não linearidade alcançada, igual a 106 para uma população de 845.712, bem superior aos cortes 0, 1 e 3.

- Notamos que apesar de termos aumentado a quantidade de gerações e elementos avaliados, a não linearidade máxima alcançada continua sendo 106.

Tabela 6.6: Resultados do Corte Aleatório em 1 Ponto de 5 Gerações e 4 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
5	0	100	2345	1050	105	0	32	104	133999
5	1	100	6197	2567	281	0	32	104	504959
5	2	100	3471	1568	202	4	32	106	320333
5	3	41	14353	5598	511	5	28	106	378527
Total		341	26366	10783	1099	9	28	106	1337818

Analisando os resultados alcançados na Tabela 6.6, podemos verificar para 5 Gerações que:

- Os cortes do tipo 0 e 1 em suas 100 rodadas apresentaram não linearidade máxima de 104.
- O corte do tipo 2 e 3 apresentaram não linearidade máxima de 106.
- O corte do tipo 3 em suas 41 rodadas apresentou uma população final um pouco superior que a 2 e alcançou a mesma não lineridade que 2.
- O corte do tipo 3 apresentou o elemento com a menor não linearidade alcançada, igual a 28.
- Notamos que apesar de termos aumentado a quantidade de gerações, a não linearidade máxima continua sendo 106, sendo que a quantidade destes elementos diminuiu, talvez em função de não termos testado as 100 vezes para o corte do tipo 3.

Na Tabela 6.7 abaixo, temos os resultados alcançados para os testes com 1 ponto de corte escolhido para todos os parâmetros especificados e com recombinação de 1 a 5 gerações.

Analisando os elementos compilados do corte aleatório de 1 ponto e 4 parâmetros na Tabela 6.7, podemos verificar que:

Tabela 6.7: Total dos Resultados do Corte Aleatório em 1 Ponto e 4 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
1	0 a 3	400	12022	6065	714	7	28	106	458701
2	0 a 3	400	22262	9974	1032	4	32	106	1138237
3	0 a 3	400	45434	19323	1866	14	32	106	1706634
4	0 a 3	400	60290	24802	2429	23	30	106	1853106
5	0 a 3	341	26366	10783	1099	9	28	106	1337818
Total	0 a 3	1941	166374	70947	7140	57	28	106	6494496
%	0 a 3		2,562	1,092	0,110	0,001			

- Todas as gerações com os 4 possíveis parâmetros alcançaram a não linearidade máxima de 106.

- A não linearidade mínima alcançada foi de 28.

- A geração 4 alcançou a maior quantidade de elementos com a máxima não linearidade alcançada.

- Para uma quantidade de $6.494.496$ foram encontrados somente 57 elementos com não linearidade 106, isto representa a complexidade em encontrar elementos com alta não linearidade.

- A percentagem de elementos com não linearidade 106 é muito pequena comparando com a quantide total de elementos gerados e avaliados.

- Para um total de $6.494.496$ de elementos, $2,562\%$ alcançaram a não linearidade de 100, dos elementos $1,092\%$ alcançaram a não linearidade de 102, dos elementos $0,110\%$ alcançaram a não linearidade de 104 e $0,001\%$ alcançaram a não linearidade de 106.

Na Figura 6.2, temos o gráfico dos resultados encontrados na Tabela 6.7.

Figura 6.2: Total Corte Aleatório em 1 Ponto com 4 Parâmetros

6.4.2 Corte Aleatório em 2 Pontos

Nas tabelas abaixo, temos os resultados alcançados para os testes com 2 pontos de corte escolhidos em função do parâmetro especificado, com recombinação de 1 geração a 5 gerações.

Os parâmetros possíveis para os testes deste grupo são o tipo de corte, que pode assumir os seguintes valores:

- 32 - 32 bits entre o primeiro ponto e segundo ponto de corte;
- 64 - 64 bits entre o primeiro ponto e segundo ponto de corte;
- 96 - 96 bits entre o primeiro ponto e segundo ponto de corte;
- 128 - 128 bits entre o primeiro ponto e segundo ponto de corte;
- 160 - 160 bits entre o primeiro ponto e segundo ponto de corte;

- 192 - 192 bits entre o primeiro ponto e segundo ponto de corte;
- 224 - 224 bits entre o primeiro ponto e segundo ponto de corte;
- 256 - 256 bits entre o primeiro ponto e segundo ponto de corte.

Tabela 6.8: Resultados do Corte Aleatório em 2 Pontos de 1 Geração e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
1	32	100	1247	706	79	0	32	104	40000
1	64	100	1429	771	108	1	30	106	40000
1	96	100	1627	839	94	0	32	104	40004
1	128	100	1807	942	111	4	32	106	40011
1	160	100	1994	1012	141	0	32	104	40018
1	192	100	2132	1059	120	1	32	106	40022
1	224	100	2273	1147	117	0	32	104	40040
1	256	100	2276	1227	152	0	32	104	40095
Total		800	14785	7703	922	6	30	106	320190

Analisando os resultados alcançados na Tabela 6.8, podemos verificar para 1 Geração que:

- Os cortes do tipo 32, 96, 160, 224 e 256 em suas 100 rodadas apresentaram não linearidade máxima de 104.
- Os cortes do tipo 64, 128 e 192 apresentaram não linearidade máxima de 106.
- O corte do tipo 128 em suas 100 rodadas apresentou a maior quantidade de elementos com a máxima não linearidade alcançada.

Analisando os resultados alcançados na Tabela 6.9, podemos verificar para 2 Gerações que:

- Os cortes do tipo 32, 96, 128, e 160 em suas 100 rodadas apresentaram não linearidade máxima de 104.
- Os cortes do tipo 64, 192, 224 e 256 apresentaram não linearidade máxima de 106.

Tabela 6.9: Resultados do Corte Aleatório em 2 Pontos de 2 Gerações e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
2	32	100	1225	718	88	0	32	104	40020
2	64	100	1414	754	96	2	32	106	41360
2	96	100	1695	844	111	0	32	104	45186
2	128	100	2114	1102	120	0	32	104	53739
2	160	100	2602	1246	143	0	32	104	61027
2	192	100	3466	1653	195	3	32	106	79785
2	224	100	4326	2023	233	5	32	106	93688
2	256	100	5643	2541	276	7	32	106	109266
Total		800	22485	10881	1262	17	32	106	524071

- O corte do tipo 256 em suas 100 rodadas apresentou a maior quantidade de elementos com a máxima não linearidade alcançada de 106 e também a maior população gerada.

Tabela 6.10: Resultados do Corte Aleatório em 2 Pontos de 3 Gerações e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
3	32	100	1161	774	95	0	34	104	40018
3	64	100	1413	780	103	0	32	104	42066
3	96	100	1725	917	102	1	32	106	49142
3	128	100	2541	1326	178	0	32	104	75967
3	160	100	5482	2301	240	0	32	104	137458
3	192	100	13571	5789	537	2	34	106	282870
3	224	100	8423	3595	300	2	34	106	92271
3	256	-	-	-	-	-	-	-	-
Total		700	34316	15482	1555	5	32	106	719792

Analisando os resultados alcançados na Tabela 6.10, podemos verificar para 3 Gerações que:

- Os cortes do tipo 32, 64, 128, 160 e 256 em suas 100 rodadas apresentaram não linearidade máxima de 104.

- Os cortes do tipo 96, 192 e 224 apresentaram não linearidade máxima de 106.

Tabela 6.11: Resultados do Corte Aleatório em 2 Pontos de 4 Gerações e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
4	32	100	1232	733	81	0	32	104	40118
4	64	100	1423	722	89	1	32	106	41066
4	96	100	1648	879	110	0	30	104	47284
4	128	100	3532	1640	189	0	32	104	91095
4	160	100	3494	1413	112	0	34	104	29241
4	192	-	-	-	-	-	-	-	-
4	224	-	-	-	-	-	-	-	-
4	256	-	-	-	-	-	-	-	-
Total		500	11329	5387	581	1	30	106	248804

- O corte do tipo 256 não pode ser testado.

Analisando os resultados alcançados na Tabela 6.11, podemos verificar para 4 Gerações que:

- Os cortes do tipo 32, 96, 128 e 160 em suas 100 rodadas apresentaram não linearidade máxima de 104.
- O corte do tipo 64 em suas 100 rodadas apresentou a não linearidade máxima de 106.
- Os cortes do tipo 192, 224 e 256 não puderam ser avaliadas.

Tabela 6.12: Resultados do Corte Aleatório em 2 Pontos de 5 Gerações e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
5	32	100	1239	711	97	1	32	106	40064
5	64	100	1542	753	82	1	32	106	41505
5	96	100	1712	941	89	0	32	104	48557
5	128	100	6199	2566	262	3	32	106	142351
5	160	100	12151	4922	421	2	32	106	160929
5	192	-	-	-	-	-	-	-	-
5	224	-	-	-	-	-	-	-	-
5	256	-	-	-	-	-	-	-	-
Total		500	22843	9893	951	7	32	106	433406

Analisando os resultados alcançados na Tabela 6.12, podemos verificar para 5 Gerações que:

- O corte do tipo 96 em suas 100 rodadas apresentou a não linearidade máxima de 104.
- Os cortes do tipo 32, 64, 128 e 160 em suas 100 rodadas apresentaram a não linearidade máxima de 106.
- Os cortes do tipo 192, 224 e 256 não puderam ser avaliadas.

Na Tabela 6.13 abaixo, temos os resultados alcançados para os testes com 2 pontos de corte escolhidos para todos os parâmetros especificados e com recombinação de 1 a 5 gerações.

Tabela 6.13: Total dos Resultados do Corte Aleatório em 2 Pontos e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
1	32 a 256	800	14785	7703	922	6	30	106	320190
2	32 a 256	800	22485	10881	1262	17	32	106	524071
3	32 a 256	700	34316	15482	1555	5	32	106	719792
4	32 a 256	500	11329	5387	581	1	30	106	248804
5	32 a 256	500	22843	9893	951	7	32	106	433406
Total		3300	105758	49346	5271	36	30	106	2246263
%			4,708	2,197	0,230	0,002			

Analisando a compilação dos resultados do corte aleatório de 2 pontos e 8 parâmetros para as 5 gerações da Tabela 6.13, podemos verificar que:

- Os 8 tipos de corte em suas 100 rodadas apresentaram a não linearidade máxima de 106.
- Os 8 tipos de corte para 2 gerações apresentaram a maior quantidade de elementos com a não linearidade máxima de 106.
- Para um total de 2.246.263 foram conseguidos somente 36 elementos com a máxima não linearidade alcançada.

- Este tipo de teste não foi finalizado em função dos prazos para entrega do projeto.
- Para um total de $2.246.263$ de elementos, $4,708\%$ alcançaram a não linearidade de 100, dos elementos $2,197\%$ alcançaram a não linearidade de 102, dos elementos $0,230\%$ alcançaram a não linearidade de 104 e $0,002\%$ alcançaram a não linearidade de 106.

Na Figura 6.3, temos a representação dos resultados encontrados na Tabela 6.13.

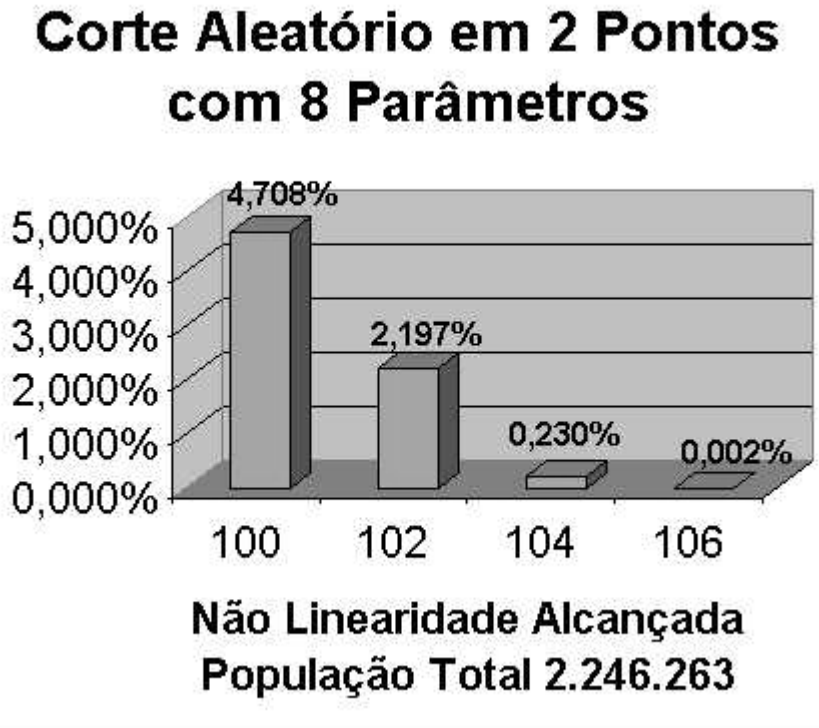

Figura 6.3: Total de Corte Aleatório em 2 Pontos com 8 Parâmetros

6.4.3 Corte Fixo em 2 Pontos

Nas tabelas abaixo temos os resultados alcançados para os testes com 2 pontos de corte, escolhidos em função do parâmetro especificado, com recombinação de 1 geração a 5 gerações.

Os parâmetros possíveis para os testes deste grupo são o tipo de corte, que pode assumir os seguintes valores:

- 1 - 256: O primeiro ponto de corte no bit 1 e o segundo no bit 256;
- 257 - 512: O primeiro ponto de corte no bit 257 e o segundo no bit 512;
- 513 - 768: O primeiro ponto de corte no bit 513 e o segundo no bit 768;
- 769 - 1024: O primeiro ponto de corte no bit 769 e o segundo no bit 1024;
- 1025 - 1280: O primeiro ponto de corte no bit 1025 e o segundo no bit 1280;
- 1281 - 1536: O primeiro ponto de corte no bit 1281 e o segundo no bit 1536;
- 1537 - 1792: O primeiro ponto de corte no bit 1537 e o segundo no bit 1792;
- 1793 - 2048: O primeiro ponto de corte no bit 1793 e o segundo no bit 2048.

Tabela 6.14: Resultados do Corte Fixo em 2 Pontos de 1 Geração e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
1	1 - 256	100	4134	2018	241	1	34	106	139900
1	257 - 512	100	4001	1956	247	2	32	106	137902
1	513 - 768	100	4254	1983	243	1	28	106	137902
1	769 -1024	100	3792	1870	247	2	30	106	135904
1	1025 - 1280	100	1206	699	95	0	32	104	40000
1	1281 - 1536	100	1210	703	86	1	32	106	40000
1	1537 - 1792	100	1186	725	89	0	32	104	40000
1	1793 - 2048	100	1179	729	92	0	32	104	40000
Total		800	20962	10683	1340	7	28	106	711608

Analisando os resultados alcançados na Tabela 6.14, podemos verificar para 1 Geração que:

- Os cortes do tipo $1025 - 1280$, $1537 - 1792$ e $1793 - 2048$ em suas 100 rodadas apresentaram não linearidade máxima de 104.
- Os demais tipos de corte apresentaram não linearidade máxima de 106.
- A quantidade de elementos com máxima não linearidade alcançada de 106 foi bem pequena para a quantidade da população gerada e avaliada.
- Os quatro primeiros tipos de corte apresentaram um melhor evolução em relação aos demais, isto pode ser notado em função da população gerada e avaliada.
- Os quatro primeiros tipos de corte foram os responsáveis pela maioria dos resultados de máxima não linearidade alcançados.

Tabela 6.15: Resultados do Corte Fixo em 2 Pontos de 2 Gerações e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
2	1 - 256	100	12864	5631	605	7	30	106	439864
2	257 - 512	100	18069	7877	816	7	32	106	408283
2	513 - 768	100	22354	9945	975	11	32	106	450123
2	769 -1024	100	11212	4781	472	15	30	106	328407
2	1025 - 1280	100	1219	707	84	0	32	104	40000
2	1281 - 1536	100	1211	707	82	0	32	104	40000
2	1537 - 1792	100	1219	686	94	1	32	106	40000
2	1793 - 2048	100	1180	730	90	0	32	104	40000
Total		800	69328	31064	3218	41	30	106	1786677

Analisando os resultados alcançados na Tabela 6.15, podemos verificar para 2 Gerações que:

- Os cortes do tipo $1025 - 1280$, $1537 - 1792$ e $1793 - 2048$ em suas 100 rodadas apresentaram não linearidade máxima de 104.

- Os demais tipos de corte apresentaram não linearidade máxima de 106.
- A quantidade de elementos com máxima não linearidade alcançada de 106 aumentou consideravelmente em relação ao teste anterior.
- Os quatro primeiros tipos de corte voltaram a apresentar uma melhor evolução em relação aos demais, isto pode ser notado em função da população gerada e avaliada.
- Os quatro primeiros tipos de corte voltaram a ser responsáveis pela maioria dos resultados de máxima não linearidade alcançados.

Tabela 6.16: Resultados do Corte Fixo em 2 Pontos de 3 Gerações e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
3	1 - 256	100	31913	13316	1149	17	30	106	309068
3	257 - 512	-	-	-	-	-	-	-	-
3	513 - 768	-	-	-	-	-	-	-	-
3	769 -1024	-	-	-	-	-	-	-	-
3	1025 - 1280	100	1158	747	95	2	32	106	40000
3	1281 - 1536	100	1173	729	98	0	32	104	40000
3	1537 - 1792	100	1237	678	86	0	32	104	40000
3	1793 - 2048	100	1181	739	80	0	32	104	40000
Total		500	36662	16209	1508	19	30	106	469068

Analisando os resultados alcançados na Tabela 6.16, podemos verificar para 3 Gerações que:

- Os cortes do tipo $1281 - 1536$, $1537 - 1792$ e $1793 - 2048$ em suas 100 rodadas apresentaram não linearidade máxima de 104.
- Os cortes do tipo $1 - 256$ e $1025 - 1280$ em suas 100 rodadas apresentaram não linearidade máxima de 106.

- Os demais testes não puderam ser realizados.

- Dos quatro primeiros tipos de corte, somente o primeiro pode ser realizado, e este foi responsável pela maioria dos resultados de máxima não linearidade alcançados.

Tabela 6.17: Resultados do Corte Fixo em 2 Pontos de 4 Gerações e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
4	1 - 256	100	27698	11044	947	11	30	106	255710
4	257 - 512	-	-	-	-	-	-	-	-
4	513 - 768	-	-	-	-	-	-	-	-
4	769 -1024	-	-	-	-	-	-	-	-
4	1025 - 1280	100	1164	736	99	1	32	106	40000
4	1281 - 1536	100	1197	709	94	0	32	104	40000
4	1537 - 1792	100	1219	708	73	0	32	104	40000
4	1793 - 2048	100	1184	716	100	0	32	104	40000
Total		500	32462	13913	1313	12	30	106	332967

Analisando os resultados alcançados na Tabela 6.17, podemos verificar para 4 Gerações que:

- Os cortes do tipo $1281 - 1536$, $1537 - 1792$ e $1793 - 2048$ em suas 100 rodadas apresentaram não linearidade máxima de 104.

- Os cortes do tipo $1 - 256$ e $1025 - 1280$ em suas 100 rodadas apresentaram não linearidade máxima de 106.

- Os demais testes não puderam ser realizados.

- Dos quatro primeiros tipos de corte, somente o primeiro pode ser realizado, e este foi responsável pela maioria dos resultados de máxima não linearidade alcançados.

Tabela 6.18: Resultados do Corte Fixo em 2 Pontos de 5 Gerações e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
5	1 - 256	100	16837	6673	571	2	32	106	172967
5	257 - 512	-	-	-	-	-	-	-	-
5	513 - 768	-	-	-	-	-	-	-	-
5	769 -1024	-	-	-	-	-	-	-	-
5	1025 - 1280	100	1212	702	86	0	32	104	40000
5	1281 - 1536	100	1188	724	87	1	32	106	40000
5	1537 - 1792	100	1184	742	73	1	32	106	40000
5	1793 - 2048	100	1213	691	96	0	32	104	40000
Total		500	21634	9532	913	4	32	106	332967

Analisando os resultados alcançados na Tabela 6.18, podemos verificar para 5 Gerações que:

- Os cortes do tipo $1025 - 1280$ e $1793 - 2048$ em suas 100 rodadas apresentaram não linearidade máxima de 104.
- Os cortes do tipo $1 - 256$, $1281 - 1536$ e $1537 - 1792$ em suas 100 rodadas apresentaram não linearidade máxima de 106.
- Os demais testes não puderam ser realizados.
- Dos quatro primeiros tipos de corte, somente o primeiro pode ser realizado, e este foi responsável pela maioria dos resultados de máxima não linearidade alcançados, mas não tão significativamente como os anteriores.

Na Tabela 6.19 abaixo, temos os resultados alcançados para os testes com 2 pontos de corte escolhido para todos os parâmetro especificados e com recombinação de 1 a 5 gerações.

Tabela 6.19: Total dos Resultados do Corte Fixo em 2 Pontos e 8 Parâmetros

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
1		800	20962	10683	1340	7	28	106	711608
2		800	69328	31064	3218	41	30	106	1786677
3		500	36662	16209	1508	19	30	106	469068
4		500	32462	13913	1313	12	30	106	415710
5		500	21634	9532	913	4	32	106	332967
Total		3100	181048	81401	8292	83	28	106	3716030
%			4,872	2,191	0,223	0,002			

Analisando os resultados compilados na Tabela 6.19 referentes aos testes com o corte fixo em 2 pontos de 1 a 5 gerações para 8 parâmetros, podemos verificar que:

- Todos as gerações apresentaram elementos com a não linearidade máxima de 106.
- Os testes realizados com 2 gerações apresentaram os melhores resultados de elementos com não linearidade máxima de 106.
- Nos testes foram encontrados elementos com não linearidade mínima de 28 e máxima de 106.
- Dos quatro primeiros tipos de corte, somente o primeiro pode ser realizado, e este foi responsável pela maioria dos resultados de máxima não linearidade alcançados, mas não tão significativamente como os anteriores.
- Este tipo de teste não foi finalizado em função dos prazos para entrega do projeto.
- Para um total de $3.716.030$ de elementos, $4,872\%$ alcançaram a não linearidade de 100, dos elementos $2,191\%$ alcançaram a não linearidade de 102, dos elementos $0,223\%$ alcançaram a não linearidade de 104 e $0,002\%$ alcançaram a não linearidade de 106.

Na Figura 6.4, temos a representação dos resultados encontrados na Tabela 6.19.

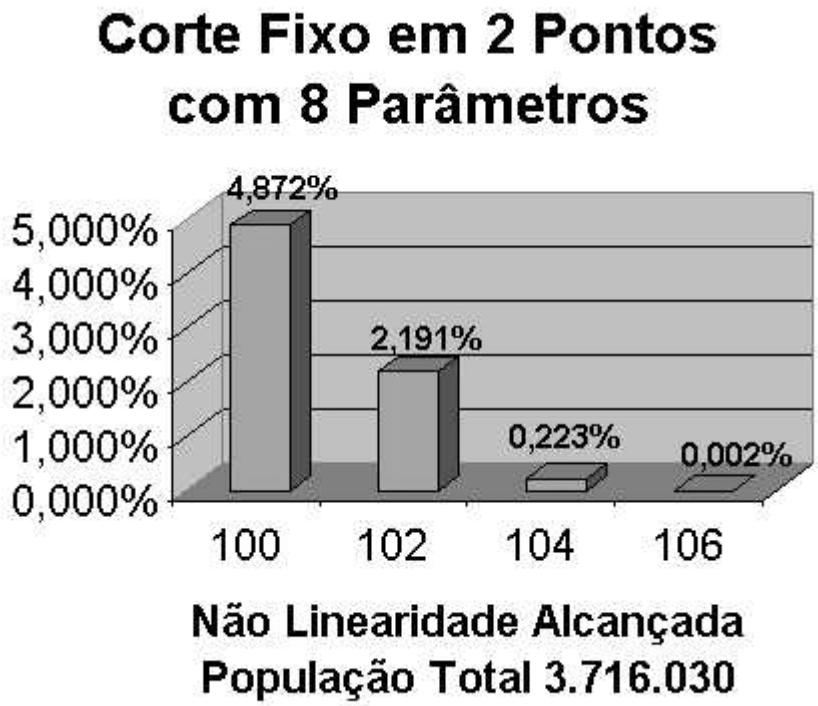

Figura 6.4: Total de Corte Fixo em 2 Pontos com 8 Parâmetros

6.4.4 Total Geral dos Resultados Realizados

Na Tabela 6.20 abaixo, temos os resultados alcançados para os os grupos de testes realizados.

Tabela 6.20: Total Geral dos Resultados com Corte Fixo e Aleatório

Geração	Corte	Rodadas	NL 100	NL 102	NL 104	NL 106	NL Mínima	NL Máxima	População
Total	Aleatório 1 Ponto	1941	166374	70947	7140	57	28	106	6494496
Total	Aleatório 2 Pontos	3300	105758	49346	5271	36	30	106	2246263
Total	Fixo 2 Pontos	3100	181048	81401	8292	83	28	106	3716030
Total Geral		8341	453180	201694	20703	176	28	106	12456789
%			3,638	1,619	0,166	0,001			

Analisando os resultados compilados das tabelas anteriores, na Tabela

6.20 referentes aos testes realizados, podemos verificar que:

- A maior população gerada, foi do corte aleatório em 1 ponto.
- O pior resultado alcançado de população e quantidade de elementos com não linearidade máxima de 106 foi o do corte aleatório em 2 pontos.
- Os testes com o corte aleatório em 2 pontos não puderam ser finalizados em função do prazo do projeto.
- O melhor resultado alcançado de elementos com não linearidade de 106 foram conseguidos pelo corte fixo em 2 pontos, apesar dos testes com 4 tipos de corte terem sido realizados somente para a 1 geração e 2 gerações.
- A mínima não linearidade alcançada foi de 28 e a máxima de 106 para um total de $12.456.789$.
- Em um total de $12.456.789$ somente 176 elementos com a máxima não linearidade foram alcançados.
- Para um total de $12.456.789$ de elementos, $3,638\%$ alcançaram a não linearidade de 100, dos elementos $1,619\%$ alcançaram a não linearidade de 102, dos elementos $0,166\%$ alcançaram a não linearidade de 104 e $0,001\%$ alcançaram a não linearidade de 106.
- Diante do número de elementos que foram gerados e avaliados, nota-se a dificuldade de encontrar elementos com alta não linearidade.

Na Figura 6.5, temos a representação dos resultados compilados na Tabela 6.20, representando percentualmente a quantidade de elementos alcançados em função de sua não linearidade.

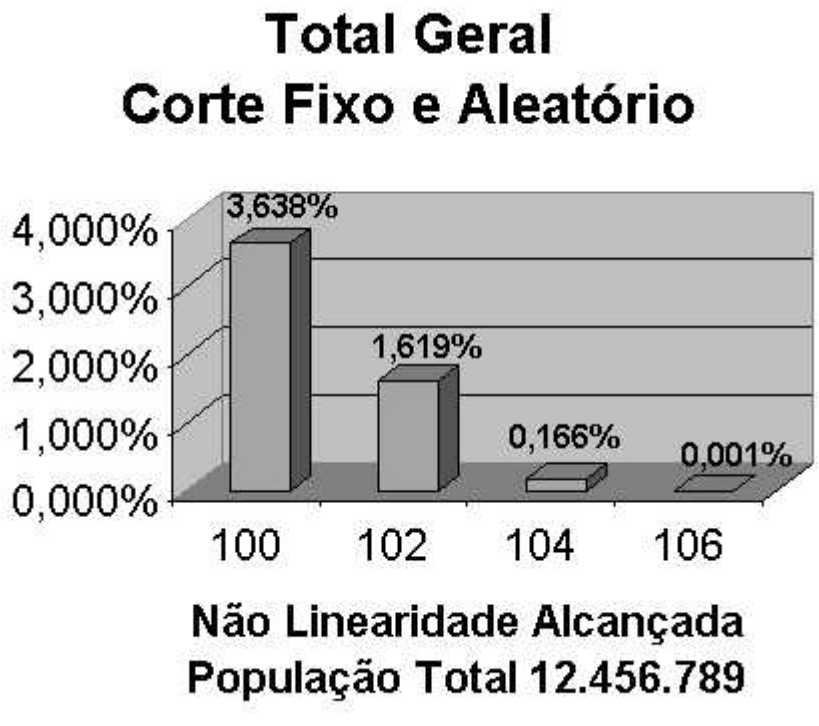

Figura 6.5: Total Geral de Corte Fixo e Aleatório

6.5 Conclusão

Este capítulo apresenta os resultados alcançados para a implementação realizada. Foram realizados três tipos de testes: Corte Aleatório, Corte Fixo em 2 Pontos e Corte Aleatório em 2 Pontos.

Apresentamos os problemas e dificuldades encontradas para a implementação e realização dos testes, em função dos prazos e indisponibilidade de laboratório para realizar os testes, os mesmos não foram testados em sua plenitude.

Com base nos resultados alcançados por tipo de testes, foram geradas tabelas e gráficos contendo os resultados.

Para um total de 12.456.789 de Caixas-S compatíveis com o AES geradas, a máxima não linearidade alcançada foi de 106, sendo que a não linearidade do AES é 112. Isto mostra a complexidade e a dificuldade que existe em projetar Caixas-S com alta não linearidade.

Capítulo 7

Considerações Finais

A revisão bibliográfica sobre o projeto e avaliação de Caixas-S e Algoritmos Genéticos foi de grande importância para o entendimento do desenvolvimento da implementação que tem por objetivo encontrar Caixas-S compatíveis com o AES, com alta não linearidade.

Conforme os objetivos iniciais dessa dissertação, pode-se tecer os seguintes comentários e conclusões:

- As Caixas-S são estruturas importantíssimas para a segurança dos algoritmos que as utilizam;
- As Caixas-S são estruturas simples em sua proposta, mas muito complexas em seu projeto;
- As Caixas-S apresentam características importantes no projeto de algoritmos de criptografia;
- A implementação utilizada nesta abordagem consome muito recurso de processamento por parte do hardware utilizado;
- A técnica utilizada neste trabalho é mais eficiente do que encontrar Caixas-S de forma aleatória, isto foi verificado analisando os resultados conhecidos da literatura com os encontrados;

- A quantidade de Caixas-S geradas e testadas nos representa a complexidade que é encontrar elementos com alta não linearidade como a utilizada pelo AES, que possui não linearidade de 112.
- Apesar da grande quantidade de Caixas-S geradas e testadas, a máxima não linearidade alcançada foi de 106.
- A implementação realizada foi sobre a Caixa-S como um todo e não sobre uma função booleana baseada na Caixa-S, como é conhecida na literatura.
- A questão da duplicação de Caixas-S foi uma preocupação abordada na implementação, ou seja, se uma Caixa-S já existente na população é gerada, esta é descartada.
- A questão da duplicação de Caixas-S não pode ser considerada para a totalidade de Caixas-S geradas, pois estas eram geradas em várias máquinas simultaneamente, sem ligação entre si, em função deste ambiente é bem possível termos contabilizado elementos duplicados.
- Este trabalho apresenta e analisa as características dos principais protocolos de criptografia simétricos em função de suas Caixas-S.
- Este trabalho apresenta e analisa as características dos principais protocolos de criptografia simétricos em função do projetos de suas Caixas-S.
- Em função dos resultados conhecidos de não linearidade de Caixas-S compatíveis com o AES, podemos concluir que os métodos baseados em Heurísticas tem conseguido excelentes resultados e podem ser considerados um método muito efetivo para esta função.
- Foi realizado uma implementação utilizando o paradigma de orintação a objetos da técnica de Algoritmos Genéticos na liguagem de programação C++.
- Foi elaborado uma implementação utilizando o paradigma de orientação a objetos do cálculo de não linearidade utilizando a Transformada Rápida de Walsh-Hadamard na linguagem de programação C++.

A partir deste trabalho, pode-se vislumbrar alguns trabalhos futuros como:

- Implementar outros critérios de avaliação para as Caixas-S geradas;
- Fazer novos testes com a implementação realizada, utilizando como população inicial as melhores Caixas-S encontradas neste trabalho.
- Fazer um estudo mais profundo da característica observada pelos resultados atingidos com o teste de corte fixo em dois pontos, principalmente com os 4 primeiros tipos de corte, com o objetivo de descobrir porque este tipo de corte apresenta melhores resultados.

Referências Bibliográficas

[ADA 90] ADAMS, C. M.; TAVARES, S. E. The use of bent sequences to achieve higher-order strict avalanche criterion in s-box design. Kingston, Ontario, Canada: Queen's University, 1990. Relatório TécnicoTR 90-013.

[ADA 93] ADAMS, C. M.; TAVARES, S. E. Designing s-boxes for ciphers resistant to differential cryptanalysis. In: PROCEEDINGS OF THE 3RD SYMPOSIUM ON STATE AND PROGRESS OF RESEARCH IN CRYPTOGRAPHY, 1993, 1993. **Proceedings...** Fondazione Ugo Bordoni: [s.n.], 1993. p.181–190.

[ADA 99] ADAMS, C.; GILCHRIST., J. **The CAST-256 Encryption Algorithm - RFC 2612**. Network Working Group, 1999. http://www.ietf.org/rfc/rfc2612.txt.

[AND 98] ANDERSON, R.; ALL., E. **Serpent: A Proposal for the Advanced Encryption Standard**. Cambridge University, Technion, University of Bergen, 1998. http:/www.cl.cam.ac.uk/ rja14/serpent.html.

[BÄC 96] BÄCK, T. **Evolutionary Algoritms in Theory and Practice**. Oxford University Press, 1996.

[BAN 99] Banzhaf, W.; Reeves, C., editors. **Foundations of Genetic Algorithms-5**. Morgan Kaufmann Publishers, Inc., 1999.

[BIT 98] BITTENCOURT, G. **Inteligência Artificial: Ferramentas e Teorias**. Florianópolis: Editora da UFSC, 1998.

[BUR 99] BURWICK, C.; ALL., E. **MARS a Candidate Cipher for AES**. IBM Corporation, 1999. http://www.research.ibm.com/security/mars.html.

[COP 94] COPPERSMITH, D. **The Data Encryption Standard and Its Strength Against Attacks**, IBM Journal of Research and Development. ed., 1994.

[COP 99] COPPERSMITH, D.; ALL., E. **The MARS Encryption Algorithm**. IBM Corporation, 1999. http://www.research.ibm.com/security/mars.html.

[DAE 99] DAEMEN, J.; RIJMEN, V. **The Rijndael Block Cipher**, 1999. http://www.nist.gov/aes.

[DAG 95] DAGHLIAN, J. **Lógica e Álgebra de Boole**. Editora Atlas, 1995.

[DAS 97] Dasgupta, D.; Michalewicz, Z., editors. **Evolutionary Algorithms in Engineering Applications**. Springer-Verlag, 1997.

[DAV 91] DAVIS, L. **Handbook of Genetic Algorithms**. New York: Van Nostrand Reinhold, 1991.

[DAW 91] DAWSON, M. H.; TAVARES, S. E. An expanded set of s-box design criteria based on information theory and its relation to differential-like attacks. 1991. **Proceedings...** Brington, UK: Spring Verlag, 1991.

[DAW 92] DAWSON, M.H. E TAVARES, S. An expanded set of s-box design criteria based on information theory an its relation to differential attacks. In: ADVANCES IN CRYPTOLOGY - EUROCRYPT´91, 1992. **Proceedings...** New York: Springer-Verlag, 1992. p.352–365.

[FEI 73] FEISTEL, H. Cryptography and computer privacy. **Scientific American**, [S.l.], v.Vol. 228, p.No. 5, 15–23, 1973.

[FER 86] FERREIRA, A. B. D. H. **Novo Dicionário Aurélio Da Lingua Portuguesa**. 2a. ed. Editora Nova Fronteira, 1986.

[FIP 77] FIPS46. **Data Encryption Standard - NBS FIPS PUB 46**. National Bureau of Standards, 1977.

[FIP 80] FIPS81. **DES Modes of Operation - NBS FIPS PUB 81**. National Bureau of Standards, 1980.

[GOL 89] GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization & Machine Learning**. Reading, Massachusetts: Addison Wesley Logman, 1989.

[HAU 98] HAUPT, R. L.; HAUPT, S. E. **Practical Genetic Algorithms**. New York: John Wiley & Sons, 1998.

[HEY 93] HEYS, H. M.; TAVARES, S. E. Substitution-permutation networks resistant to differential an linear cryptanalysis. Kingston, Ontario, Canada: Queen's University, 1993. Relatório técnico.

[HEY 94] HEYS, H. M.; TAVARES, S. E. Substitution-permutation networks resistant to differential and linear cryptanalysis. Queen's University, August, 1994. Relatório técnico.

[JS 94a] J. SEBERRY, X. M. Z.; ZHENG, Y. Nonlinearly balanced boolean functions and their propagation characteristics. In: LNCS, V. ., editor, ADVANCES IN CRYPTOLOGY - CRYPTO '93, 1994. Springer-Verlag, 1994. p.49–60.

[JS 94b] JENNIFER SEBERRY, X. Z.; ZHENG, Y. Relationships among nonlinearity criteria. 1994. **Proceedings...** Perugia, Italy: Spring-Verlag, 1994.

[KAM 79] KAM, J. E. G. D. Structured design of substitution-permutation encryption networks. **IEEE Transactions on Computers**, [S.l.], v.C-28(10):747-753, 1979.

[KEL 97] KELIHER, L. **Substitution-Permutation Network Cryptosystems Using Key-Dependent S-Boxes**. Queen's University, 1997. Dissertação de Mestrado.

[KIM 90] KIM, K. **A Study on the Construction and Analysis of Substitution Boxes for Symmetric Cryptosystems**. Kingston, Ontario, Canada: Yokohama National University, 1990. Tese de Doutorado.

[LAC 99] LACERDA, E. G. M.; DE CARVALHO, A. C. P. L. F. Introdução aos algoritmos genéticos. In: ANAIS DO XIX CONGRESSO NACIONAL DA SOCIEDADE BRASILEIRA DE COMPUTACAO, 1999. **Proceedings...** PUC-Rio: [s.n.], 1999.

[LB 00] L. BURNETT, G. CARTER, E. D.; MILLAN, W. Efficient methods for generating MARS-like s-boxes. 2000. **Proceedings...** New York: Por Aparecer., 2000.

[LIN 98] LINT, J. H. V. **Introduction to Coding Theory**. 3a Edição. ed. Springer-Verlag Berling Heildelberg, 1998.

[MAT 93] MATTEWS, R. A. J. The use of genetic algorithm in cryptanalysis. 1993. [s.n.], 1993.

[MEI 90] MEIER, W.; STAFFELBACH, O. Nonlinearity criteria for cryptographic functions. 1990. Spring-Verlag, 1990. LNCS.

[MEN 97] MENEZES, A.; OORSHCOT, P.; VANSTONE, S. **Handbook of Applied Cryptography**. Boca Raton, FL: CRC Press, 1997.

[MIS 96] MISTER, S.; ADAMS, C. Practical s-box design. In: SAC'96, editor, WORKSHOP RECORD - SAC'96, 1996. **Proceedings...** Ontario, Canada: Workshop on Selected Areas in Cryptology, 1996. p.61–76.

[MIT 97] MITCHELL, M. **An Introduction to Genetic Algorithms**. Cambridge, Massachusetts, London, England: The MIT Press, 1997.

[MOS 95] MOSCATO, P.; LAGUNA, M. **Algoritmos Genéticos**, 1995. http://www.densis.fee.unicamp.br/ moscato.

[NAI 99] NAI. **An Introduction to Cryptography**. Network Associates, Inc., 1999. http://www.pgpi.org/doc/guide/6.5/en/intro/.

[NIS 00] NIST. **Advanced Encryption Standard Develpment Effort**. U.S. Commerce Department's Technology Administration, 2000. http://csrc.nist.gov/encryption/aes/.

[NYB 92] NYBERG, K. On the construction of highly nonlinear permuations. In: Eurocrypt'92, editor, ADVANCES IN CRYPTOLOGY, 1992. **Proceedings...** Berlin: Springer-Verlag, 1992. p.92–98.

[NYB 94] NYBERG, K. Differentially uniform mappings for cryptography. In: Helleseth, T., editor, ADVANCES IN CRYPTOLOGY, 1994. **Proceedings...** T. Helleseth: Springer-Verlag, 1994. p.55–64.

[RIT 98a] RITTER, T. **Active Boolean Function Nonlinearity Measurement in JavaScript**, 1998. http://www.ciphersbyritter.com.

[RIT 98b] RITTER, T. **Measured Boolean Function Nonlinearity in Variable Size Block Ciphers**, 1998. http://www.ciphersbyritter.com.

[RIT 98c] RITTER, T. **Measuring Boolean Function Nonlinearity by Walsh Transform**, 1998. http://www.ciphersbyritter.com.

[RIT 99] RITTER, T. **S-Box Design: A Literature Survey**, 1999. http://www.ciphersbyritter.com.

[ROB 95] ROBSHAW, M. J. B. Block ciphers - RSA laboratories technical report TR-601. RSA Laboratories, 1995. Relatório técnico.

[SCH 96] SCHNEIER, B. **Applied Cryptography: Protocols, Algorithms, and Source Code in**. Segunda Edição. ed. John Wiley & Sons., 1996.

[SCH 98] SCHNEIER, B. **Twofish: A 128-Bit Block Cipher**. Counterpane Internet Security, Inc., Junho, 1998. http://www.counterpane.com/twofish.html.

[SCH 00] SCHNEIER, B. **Secrets and Lies: Digital Security in a Networked World**. John Wiley & Sons, Inc., 2000.

[SMI 94] SMITH, R. E. **SGA-C: A C-Language Implementation of a Simple Genetic Algorithm**, 1994. http://www.aic.nrl.navy.mil/galist/src/.

[SPI 93] SPILLMAN, R. Cryptanalysis of knapsack cipher using genetic algorithm. 1993. [s.n.], 1993.

[STA 99] STALLINGS, W. **Cryptography And Network Security: Principles and Practice**. Segunda Edição. ed. Upper Saddle River, New Jersey: Prentice-Hall, 1999.

[TRA 01] TRAPPE, W.; WASHINGTON, L. C. **Introduction to Cryptography with Coding Theory**. Prentice Hall, 2001.

[WEB 86] WEBSTER, A. E TAVARES, S. On the design of s-box. In: ADVANCES IN CRYPTOLOGY - CRYPTO´85, 1986. **Proceedings...** New York: Springer-Verlag, 1986. p.523–534.

[WM 97] WILLIAM MILLAN, A. C.; DAWSON, E. An effective genetic algorithm for finding highly nonlinear boolean functions. 1997. **Proceedings...** Sydney: Spring Verlag, 1997.

[WM 98] WILLIAM MILLAN, A. C.; DAWSON, E. Heuristic design of cryptographically strong balanced boolean functions. 1998. Spring Verlag, 1998.

[WM 99] W. MILLAN, L. BURNETT, G. C. A. C.; DAWSON, E. Evolutionary heuristics for finding cryptographycally strong s-boxes. 1999. **Proceedings...** Sydney: Spring-Verlag, 1999.

[YOU 97] YOUSSEF, A. M. **Analysis and Design of Block Ciphers**. Kingston, Ontario, Canada: Queen's University, 1997. Tese de Doutorado.

Apêndice A

Anexo A - Caixas-S Geradas

Neste apêndice econtram-se alguns exemplos de Caixas-S geradas utilizando o método de Algoritmos Genéticos deste projeto que alcançaram a não linearidade de 106.

CAIXA-S 1

96	208	144	176	64	0	16	128	255	48	240	192	224	117	222	187
203	186	152	99	151	82	150	202	149	254	132	253	170	98	238	116
112	169	221	80	185	81	32	201	65	79	200	78	39	237	220	168
184	53	131	38	183	219	218	37	97	252	52	217	182	95	216	51
63	77	215	130	160	148	13	181	50	36	115	49	167	24	94	93
114	92	12	62	147	113	146	91	166	145	199	47	180	179	251	250
178	76	75	74	143	142	11	35	198	249	248	247	61	23	214	236
10	111	60	110	90	34	89	177	9	165	22	141	235	164	213	8
246	212	59	197	58	7	163	21	234	33	46	140	233	31	129	245
57	45	139	138	20	44	196	127	195	175	244	30	126	174	19	125
73	232	162	124	109	211	194	108	161	72	29	159	193	231	158	107
123	28	106	105	210	230	6	43	243	137	136	71	56	209	88	5
42	55	27	70	191	87	122	104	4	157	69	156	121	120	41	135
229	3	134	133	228	68	207	86	227	190	189	206	242	67	226	173
2	241	103	119	155	1	172	154	54	40	18	17	85	239	205	102
84	153	66	15	225	101	100	223	171	14	188	26	204	83	25	118

CAIXA-S 2

48	176	128	222	240	144	64	192	0	205	221	32	208	16	160	80
224	138	255	153	96	185	220	254	65	253	28	137	136	184	204	183
82	81	166	63	62	135	182	203	239	97	61	202	238	201	134	200

181	27	112	26	13	133	152	108	25	120	151	107	95	94	106	24
45	180	105	252	199	12	165	44	119	43	42	237	236	60	23	132
219	11	164	79	150	59	78	218	149	10	235	131	217	58	57	216
198	215	251	148	93	56	234	214	147	130	92	118	41	146	40	117
91	77	197	163	179	116	145	39	178	55	213	22	104	233	38	129
250	115	114	76	9	196	212	127	21	162	8	249	126	125	232	7
124	54	177	195	37	6	248	5	36	113	20	53	211	175	231	90
247	174	230	75	143	19	173	210	123	4	229	18	246	194	172	52
111	228	89	17	103	227	88	193	87	161	209	15	51	159	35	74
73	86	171	34	207	226	85	245	191	122	170	142	33	141	140	50
102	190	158	110	3	244	72	169	31	225	189	49	168	223	30	101
14	71	70	188	187	100	157	84	69	243	29	68	67	47	206	83
186	99	121	156	155	109	2	139	242	66	167	46	98	241	1	154

CAIXA-S 3

165	238	146	154	89	139	216	183	122	254	54	178	132	42	26	22
107	127	60	134	209	102	119	70	125	223	217	9	104	246	186	218
34	129	52	86	47	29	80	94	112	251	199	10	121	63	36	85
210	31	28	231	241	225	205	44	116	175	11	57	157	155	81	143
25	141	250	7	207	221	242	167	153	191	206	208	137	32	249	198
197	65	84	188	16	27	213	75	103	71	0	248	245	142	229	156
5	201	106	101	162	66	226	88	98	179	133	232	56	190	228	110
148	96	114	240	147	159	118	19	244	150	87	117	41	164	46	140
239	92	39	128	247	252	220	131	136	105	83	219	253	35	4	21
169	212	3	108	68	49	123	176	12	90	255	40	135	152	23	99
91	79	158	144	76	215	14	77	227	6	67	45	173	149	61	189
43	100	236	166	145	202	51	130	24	187	72	181	58	55	234	30
69	2	195	18	203	78	161	222	62	97	64	163	37	170	182	82
115	200	235	204	120	74	8	93	160	113	138	214	48	73	15	20
237	111	13	33	180	184	185	224	53	124	17	230	172	193	59	126
174	95	1	243	192	151	211	233	194	50	109	171	177	168	196	38

CAIXA-S 4

113	31	100	147	63	7	248	51	140	218	146	114	231	189	79	73
121	2	203	222	169	149	215	39	224	112	103	123	72	15	3	115
111	4	87	27	204	48	76	16	175	131	162	99	244	55	136	89
71	133	36	5	241	35	177	172	214	253	40	127	229	167	247	245
32	95	210	153	208	180	223	228	52	242	151	139	249	250	135	227
181	194	129	70	221	145	124	216	68	182	107	158	178	28	50	80
66	105	128	192	126	141	0	163	83	19	188	6	205	62	179	185
21	44	199	160	166	206	12	18	75	34	236	11	196	93	102	60
209	30	213	17	24	144	254	86	176	195	85	226	59	20	14	125
109	150	26	240	61	110	137	8	243	154	74	234	122	193	238	148

47	57	119	190	108	233	191	230	152	239	53	92	186	78	232	42
174	138	77	161	155	211	134	84	201	235	183	45	91	184	54	225
187	246	64	116	46	132	159	88	207	69	37	157	101	41	104	198
251	1	237	143	255	200	98	58	220	118	202	94	97	142	9	10
23	25	117	33	22	65	170	219	43	165	212	13	173	90	56	81
29	120	197	252	164	106	82	49	217	67	171	168	96	38	130	156

CAIXA-S 5

92	65	64	105	235	34	59	229	113	225	75	16	37	108	247	99
103	47	202	73	137	126	178	79	125	210	115	171	188	144	69	44
48	166	176	128	184	201	93	123	3	117	237	90	167	39	182	26
61	4	12	181	41	211	107	42	141	221	53	70	245	175	224	13
119	179	6	173	200	228	253	31	78	177	68	0	98	174	223	190
51	249	71	109	222	142	97	111	36	28	232	145	91	241	110	199
118	250	22	209	74	82	146	183	140	15	206	60	72	138	120	134
77	208	187	85	8	19	239	240	159	45	165	80	170	148	156	151
154	194	101	23	130	18	163	83	169	40	87	56	244	14	52	2
96	62	9	54	100	29	114	24	191	124	76	33	215	135	147	195
122	121	95	32	132	233	27	214	204	246	63	234	189	172	57	227
152	243	139	217	94	231	50	35	136	158	10	251	67	196	212	133
255	219	161	252	162	38	198	236	185	129	216	197	254	168	1	149
102	49	220	131	205	21	157	248	43	84	186	88	180	66	192	7
20	86	155	160	143	127	46	242	58	55	207	11	112	150	238	104
81	30	230	153	203	193	226	164	5	213	116	17	218	25	89	106

CAIXA-S 6

145	175	116	122	86	36	8	39	85	88	22	60	155	53	180	4
114	138	26	208	172	34	163	185	27	182	142	57	20	130	109	135
134	77	99	162	212	128	251	159	221	63	44	234	67	170	144	66
235	224	165	30	75	193	9	11	161	121	16	168	103	80	43	5
38	229	176	233	96	196	177	123	222	2	218	244	117	119	220	100
246	243	83	219	59	94	115	32	101	48	181	198	33	51	236	93
91	12	237	28	10	81	151	136	230	102	0	56	132	125	54	190
148	73	153	13	143	41	152	206	228	127	55	227	21	204	240	164
241	79	209	106	238	19	201	89	169	90	6	189	225	69	133	40
129	7	98	239	112	46	200	179	207	87	141	104	254	37	52	14
31	70	108	226	76	95	29	64	18	50	160	247	82	186	205	97
213	74	252	105	202	146	194	249	24	140	199	3	17	232	203	255
1	23	35	156	150	211	197	78	49	242	118	217	248	65	61	215
71	131	126	147	92	171	42	45	231	157	183	178	15	72	68	184
107	192	214	137	47	58	167	216	120	111	210	245	139	110	195	62
250	84	191	154	149	166	187	158	188	113	124	173	223	174	25	253

Apêndice B

Anexo B - Fontes

Neste apêndice econtram-se os fontes do programa de computador utilizados para implementar os algoritmos genéticos deste trabalho.

B.1 Classes

B.1.1 Classe cromossomo.h

Classe que representa o cromossomo que compora um indivíduo da população. É constituída por um conjunto de genes, de acordo com a definiç ao de TAMCROMO, e por um genoma, que é a sequência dos genes que compõe o cromossomo.

No modelo AES, a caixa S e formada por uma matriz de 16 X 16 elementos, de 8 bits cada um, o número de genes do cromossomo será de 256, sendo cada um formado por 8 bits. Isso pois a matriz original devera ser analizada como um elemento da população. Os ”defines”a seguir representam este conceito.

```
#define TAMCROMO 256

/* No modelo DES, o numero de alelos, cada locus do cromossomo,
   eh igual a 64 (16 com 4 bits).
   No modelo AES, o numero de alelos serah de 256 genes com
   8 bits cada um, ou seja, 2048 bits.
   Os 'defines' a seguir representam este conceito.
*/

#define TAMFULL  2048
```

```
#if !defined(Cromo_h)
#define Cromo_h

class gene;

class cromossomo {
    private:
        gene mGene[TAMCROMO];
        char genoma[TAMFULL];
        int matriz[TAMCROMO];

    public:
        cromossomo();
        cromossomo(cromossomo *cromo);
        cromossomo(gene gen[TAMCROMO]);
        ~cromossomo();

        bool IsEmpty();
        bool operator==(cromossomo *outroCromo);

        /*
         * Compara 2 cromossomos para verficar se sao iguais,
         * retorna true caso sejam.
        */
        bool comparaGenes(cromossomo *outroCromo);

        /* para reproducao. atribui valor para o gene do cromossomo*/
        void setGene(int index, int valor);

        gene* getGene(int index);
        void setValorGenes(gene valores[TAMCROMO]);
        int  getValorGenes(int index);

        /*metodos para manipulacao do genoma*/
        char& getGenoma();

        /*Retorna um caractere do genoma, de acordo com o indice.*/
        char getLocusFromGenoma(int locus);

        /*Atribui valor ao locus do genoma indicado pelo indice.*/
        void setLocusFromGenoma(int locus, char valor);
        void setGenoma(char genoma[TAMFULL]);

        /*metodos de conversao*/
        void GenomaFromGenes();
        void GenesFromGenoma();

        /* Metodo para atribuir valor decimal para os genes
         * deste cromossomo*/
        void valorizaGenes();

        /* Metodo que efetua a mutacao de genes repetidos.
         * Para chamar este metodo eh necessario que o cromossomo
         * esteja com seus genes atualizados de acordo com o genoma
         * e este de acordo com os valores daqueles */
        void mutacao();

        /* Criar genes aleatoriamente.*/
        void criaGenesAleatorios();
        int  numeroAleatorio();
```

```
};
#endif
```

B.1.2 Classe gene.h

```
/*
Classe que representa os genes que comporao o cromossomo. Cada
gene e constituido de um valor e de sua representação binária,
podendo o numero de bits desta variar de acordo com a definicao do
NUMLOCUS, ou seja, numero de locus do gene.
*/

#if !defined(Gene_h)
#define Gene_h

#include <stdlib.h>
#include <stdio.h>

#define NUMLOCUS 8  // 8 bits para AES

class gene {
    private:
        int valor;
        //conterah o valor em binarios que representarah o
        // gene propriamente dito e serah
        //usado para as operacoes.

        char alelo[NUMLOCUS];

    /* Metodos para acesso ao atributo valor deste gene*/

    public:
        gene();
        gene(int valor);
        ~gene();
        FILE *log;

        bool IsEmpty();
        void setValor(int valor);
        int getValor();
        bool operator==(int valor);
        void operator=(int valor);
        char getAleloFromIndex(int index);
        void setAlelo(char string[NUMLOCUS]);
        void setAleloFromIndex(int index, char valor);
        void valorToAlelo();
        /* Converte o valor dos alelos para um valor decimal
         * e atribui ao atributo 'valor' */
        void aleloToValor();
};
#endif
```

B.1.3 Classe individuo.h

```
/*

Classe que representa um individuo da populacao. Sera onstituido
de um cromossomo, cujo tamanho dependera das definicoes de
TAMCROMO e TAMFULL. Tambem sera formado por uma aptidao, uma
nao-linearidade e um valor para a funcao objetivo do problema.

*/

#if !defined(Indiv_h)
#define Indiv_h

#include <stdio.h>

class cromossomo; class individuo {
    private:
        int aptidao;
        int naoLinearidade;
        int valorFuncObj;
        cromossomo *cromo;  //tera 256 genes


    public:
        individuo();
        individuo(cromossomo *cromo);
        ~individuo();

        void setAptidao(int valor);
        void setValorFuncObj(int valor);
        int getAptidao();
        int getNaoLinearidade();
        int getValorFuncObj();
        bool IsEmpty();
        cromossomo * getCromo();

        void setGeneCromossomo(int Index, int valor);
        void setCromossomo(cromossomo *cromo);
        int getValorAleloFromIndex(int index);
        void calculaNaoLinearidade();
        /* Criar o cromossomo aleaoriamente.*/
        void criaCromoAleatorio();
        /* Imprime individuo no arquivo do parametro.*/
        void downloadIndiv(FILE *arquivo);
};

#endif
```

B.1.4 Classe lista.h

```
// lista.h: interface for the lista class.

#if !defined(LISTA_H)
#define LISTA_H

class individuo;

class lista { private:
```

```
    individuo *indiv;
    lista *next;

public:
    lista();
    virtual ~lista();

    void setNext(lista *next);
    lista* getNext();
    individuo* getIndividuo();
    void setIndividuo(individuo* indiv);
};

#endif // !defined(LISTA_H_)
```

B.1.5 Classe listaMng.h

```
// listaMng.h: interface for the listaMng class.

#if
!defined(AFX_LISTAMNG_H__54A42A21_0F30_11D6_8718_FFFFFF000000__INCLUDED_)
#define
AFX_LISTAMNG_H__54A42A21_0F30_11D6_8718_FFFFFF000000__INCLUDED_

#if _MSC_VER > 1000
    #pragma once
#endif // _MSC_VER > 1000

class individuo; class lista;

class listaMng { private:
    lista *inicio;
    lista *principal;
    lista *aux;
    int quantidadeIndiv;
public:
    listaMng();
    virtual ~listaMng();

    void incluir(individuo *indiv);
    individuo *getIndividuo(int indice);

    void setQtde(int valor);
    int getQtde();

};

#endif
//!defined(AFX_LISTAMNG_H__54A42A21_0F30_11D6_8718_FFFFFF000000__INCLUDED_)
```

B.1.6 Classe populacao.h

```
/*
```

```
Classe que representa o conjunto de individuos candidatos a
resolucao do problema proposto. Sera constituida por um conjunto
de individuos, cujo tamanho dependerah da definicao de TAMPOP.
Sera responsavel por gerenciar a escrita do arquivos que conterah
todas as populacoes que surgiram na tentativa de solucionar o
problema proposto.
*/

#include <stdio.h>
#include <stdlib.h>

#define TAMPOP 8
#define TAMCROMO 256
#define TAMFULL 2048

//class reproducao;
#if !defined(Pop_h)
#define Pop_h

class individuo; class listaMng;

class populacao {
    private:
        FILE *arquivo;
        listaMng *listaPovo;
        int geracao;
        int tamanhoPopulacao;
        int indicePopulacao;
        double mediaNL_2;
        double mediaNL_2_4;
        double mediaNL_4;
        int maxNL;
        int minNL;
        int accMaxNL;
        int accMinNL;
        int tipoXcorte;


    //reproducao mReproducao;

 public:
    populacao();

     /*Constructor que faz a reproducao da populacao anterior.
      *Informar o tipo de corte do Xover.*/

     //populacao &popAnterior
     populacao(int geracaoAnt, int tipoCorte = 0, int p1 = 0, int p2 = 0);
     ~populacao();
     void setGeracao(int valor);
     int getGeracao();
     void setTamanhoPopulacao(int tamPop);
     int getTamanhoPopulacao();
     individuo* getIndividuo(int index);

     bool loadPovo(char *entrada ); //leitura do arquivo com as matrizes.
     bool downloadPovo(char *saida); //todos para arquivo.
     bool download(char *saida = NULL);
     bool comparaPool(individuo *indiv);

     bool operator==(populacao &pop);
```

```
     populacao& operator=(populacao &pop);
     void setPovo(individuo *newIndividuo);
     bool IsEmpty();
     /* Metodo que compara todos os individuos da
        populacao com o individuo do parametro e
        retorna o indice do individuo semelhante,
        caso sejam todos diferentes, retorna -1
     */
     int comparaIndividuos(individuo* pessoa);
     /* Metodo que procura individuos semelhantes a
        partir da posicao especificada
        populacao com o individuo do parametro, o indice de partida
        e retorna retorna -1 caso nao encontre.
     */
     bool procuraIndividuos(individuo* pessoa, int indice);

     listaMng* getLista();
     /* Geracao de numero aleatorio entre 0 e 15 para ocorrencia do Xover */
     int pontoXover(int limiteSup = TAMFULL);
     /* Reproducao nao estah sendo utilizado.*/
     populacao* reproduz( );

     /* Estatisticas*/
     /* Calcula as medias de Nao-Linearidade dos individuos da populacao
      * medias das NL menores e igauis a 2, entre 2 e 4 e maiores e iguais a 4
     */
     void calcMediasNL();
     double getmediaNL_2();
     double getmediaNL_2_4();
     double getmediaNL_4();
     int getmaxNL();

     /* Impressao tmp.*/
     void relCabecalho();
     void relIndividuos(int indiceIndiv = 0, individuo * indiv = NULL);
     void relEstatisticas();
     /* O parametro deve ser um ponteiro de arquivo aberto.*/
        individuo* leIndivPopAnt(FILE *arquivo);
};

#endif
```

B.1.7 Classe fileMng.h

```
// fileMng.h: interface for the fileMng class.

#if !defined(FILEMNG_H)
#define FILEMNG_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class fileMng { public:
    fileMng();
    virtual ~fileMng();
```

```
};

#endif // !defined(FILEMNG_H)
```

B.2 Programas

B.2.1 Programa cromossomo.cpp

```
#include "stdafx.h"
#include "gene.h"
#include "cromossomo.h"

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

cromossomo::cromossomo() {
    /*codigo para geracao randomica pode ir aqui*/
    for(int k = 0; k<TAMCROMO; k++)
        matriz[k] = 0;
}

cromossomo::cromossomo(cromossomo *cromo) {
    for(int k = 0; k<TAMCROMO; k++)
        matriz[k] = 0;

    for(int i=0;i<TAMCROMO;i++)
    {
        this->mGene[i] = 0; //zerando os genes
        this->mGene[i] = cromo->mGene[i]; //atribuindo os valores
    }
} cromossomo::cromossomo(gene gen[TAMCROMO]) {
    for(int k = 0; k<TAMCROMO; k++)
        matriz[k] = 0;
    for(int i=0;i<TAMCROMO;i++)
    {
        this->mGene[i] = 0; //zerando os genes
        this->mGene[i] = gen[i]; //atribuindo os valores
    }
}

cromossomo::~cromossomo() { }

bool cromossomo::IsEmpty() {
    for(int i=0;i<TAMCROMO;i++)
    {
        return(this->mGene[i].IsEmpty());
    }
    return false;
}

bool cromossomo::operator==(cromossomo *outroCromo) {
    for(int i=0;i<TAMCROMO;i++)
    {
```

```
     if(!(this->mGene[i]== outroCromo->getGene(i)->getValor()))
        return false;
    }
    return true;
}

bool cromossomo::comparaGenes(cromossomo *outroCromo) {
    for(int i=0;i<TAMCROMO;i++)
    {
     if((this->mGene[i].getValor() != outroCromo->getGene(i)->getValor()))
         return false;
    }
    return true;
}

void cromossomo::setGene(int index, int valor)
/* para reproducao.
atribui valor para o gene do cromossomo*/
{
    this->mGene[index] = valor;
}

gene* cromossomo::getGene(int index) {
    return &(this->mGene[index]);
}

    if(this == &outroCromo)
        return *this;

    for(int i=0;i<=TAMCROMO;i++)
    {
        this->mGene[i] = outroCromo.getGene(i)->getValor();
    }
    return *this;
} */

void cromossomo::setValorGenes(gene valores[TAMCROMO]) {
    for(int i=0;i<=TAMCROMO;i++)
    {
        this->mGene[i] = valores[i];
    }
}

int  cromossomo::getValorGenes(int index) {
    return (mGene[index].getValor());
}

char& cromossomo::getGenoma() {
    return genoma[TAMFULL];
}

void cromossomo::setGenoma(char genoma[TAMFULL]) {
    for(int i = 0 ; i<TAMFULL; i++)
    {
        this->genoma[i] = genoma[i];
    }
}
/*metodos de conversao, dos valores individuais dos genes para
um cromossomo full*/

void cromossomo::GenomaFromGenes() {
```

```
    int index = 0;

    for(int j = 0; j<TAMCROMO; j++)
    {
        for(int k = 0; k<4; k++)
        {
            genoma[index] = mGene[j].getAleloFromIndex(k);
            index++;
        }
    }
}
/*metodos de conversao do valor do genoma para os valores dos
genes individuais*/

void cromossomo::GenesFromGenoma() {
    int index = 0;

    for(int j = 0; j<TAMCROMO; j++)
    {
        for(int k = 0; k<4; k++)
        {
            mGene[j].setAleloFromIndex(k,this->genoma[index]);
            index++;
        }
        /* atualiza o valor decimal do gene*/
        mGene[j].aleloToValor();
    }
}

char cromossomo::getLocusFromGenoma(int locus) {
    return (genoma[locus]);
}

void cromossomo::setLocusFromGenoma(int locus, char valor) {
    this->genoma[locus] = valor;
}

/* Metodo para atribuir valor decimal para os genes deste
cromossomo*/

void cromossomo::valorizaGenes() {
    for(int i = 0; i<TAMCROMO; i++)
    {
        this->mGene[i].aleloToValor();
    }
}

/* Metodo que efetua a mutacao de genes repetidos*/

void cromossomo::mutacao() {
    int
        val = 0, f = 0, b = 0, repetido = 0, qtd1s = 0;

    for(int k = 0; k<TAMCROMO; k++)
    {
        matriz[k] = 0;
    }

    for(int i = 0; i<TAMCROMO; i++)
    {
        /* Encontra os numeros que nao estao presentes neste cromossomo*/
```

```
        val = mGene[i].getValor();

        if(matriz[val] == 0)
        {
            matriz[val] = 1;
            qtd1s ++;
        }
    }

    i = 0;
    while(qtd1s<TAMCROMO)
    {
        val = this->mGene[i].getValor();
        for(int j = 0; j<TAMCROMO; j++)
        {
            if(val == this->mGene[j].getValor())
            {
                repetido++;
                if(repetido >=2)
                {
                    k=TAMCROMO;
                    do
                    {
                        k--;
                    }while(matriz[k] != 0);

                    this->mGene[j].setValor(k);
                    this->mGene[j].valorToAlelo();
                    matriz[k] = 1;
                    qtd1s++;
                }
            }
        }
        repetido = 0;
        i++;
    }
    this->GenomaFromGenes();
}

/* Criar genes aleatoriamente.*/

void cromossomo::criaGenesAleatorios() {
    int valor = 0, j = 0;

    for(int k = 0; k<TAMCROMO; k++)
        matriz[k] = 0;

    do
    {
        valor = this->numeroAleatorio();
        /* Verifica se o valor jah existe no cromossomo, se
         * existir, descarta-o .
         */
        if(j==0)
        {
            this->mGene[j].setValor(valor);
            this->mGene[j].valorToAlelo();
            matriz[valor] = 1;
            j++;
        }
        else
```

```
        {
            if(matriz[valor] == 0)
            {
                this->mGene[j].setValor(valor);
                this->mGene[j].valorToAlelo();
                matriz[valor] = 1;
                j++;
            }
        }
    }while(j<TAMCROMO);

    this->mutacao();

    this->GenomaFromGenes();

}

int  cromossomo::numeroAleatorio() {
    time_t t;
    int valor;

    srand((unsigned) time(&t));

    do
    {
        valor = rand()%TAMCROMO;
    }while(matriz[valor] == 1);

    return(valor);
}
```

B.2.2 Programa gene.cpp

```
#include "stdafx.h"
#include "gene.h"
#include <stdlib.h>

#include <math.h>

gene::gene() {
    valor = -1;
}

gene::gene(int valor) {
    this->valor = valor;
}

gene::~gene() { }

bool gene::IsEmpty() {
    if(valor == -1)
        return false;
    return true;
}

void gene::operator=(int valor) {
    this->valor = valor;
```

```
}

void gene::setValor(int valor) {
    this->valor = valor;
}

int gene::getValor() {
    return valor;
} /*return allele == g.allele;*/

bool gene::operator==(int valor) {
    return (this->valor == valor);
}

char gene::getAleloFromIndex(int index) {
    return alelo[index];
}

void gene::setAlelo(char string[NUMLOCUS]) {
    for(int i =0;i<NUMLOCUS;i++)
        alelo[i] = string[i];
}

void gene::valorToAlelo() {
    char buffer[2];
    int bitPos = (int) pow(2, NUMLOCUS -1);
    int numLocus = NUMLOCUS - 1;

    for(int i =0;i<NUMLOCUS;i++)
    {
        _itoa((valor & bitPos)>>(numLocus - i), buffer, 10);
        alelo[i] = buffer[0];
        bitPos = bitPos >> 1; //shiftando para filtrar o proximo bit.
    }
}

void gene::aleloToValor() {
    int temp[NUMLOCUS];
    char buffer[2];
    int bitPos = (int) pow(2, NUMLOCUS -1);
    int numLocus = NUMLOCUS - 1;

    for(int i = 0; i < NUMLOCUS; i++)
    {
        buffer[0] = alelo[i];
        temp[i] = atoi(buffer);
        buffer[0] = '0';
    }
    //verifica se o valor eh -1, se for,
    //iguala a zero para poder realizar o 'OR'...

        valor = 0;

    for(int k =0;k < NUMLOCUS; k++)
    {
        valor = valor | (temp[k]) << (numLocus - k);
    }
}

void gene::setAleloFromIndex(int index, char valor) {
    this->alelo[index] = valor;
```

```
}
```

B.2.3 Programa individuo.cpp

```
#include "stdafx.h"

#include "gene.h"
#include "cromossomo.h"
#include "individuo.h"
#include <math.h>


individuo::individuo() {
    cromo = new cromossomo();
}
individuo::individuo(cromossomo *cromo) {
    cromo = new cromossomo(cromo);
}

individuo::~individuo() {

}

void individuo::setAptidao(int valor) {
    this->aptidao = valor;
}

void individuo::setValorFuncObj(int valor) {
    this->valorFuncObj = valor;
}

int individuo::getAptidao() {
    return aptidao;
}

int individuo::getNaoLinearidade() {
    return naoLinearidade;
}

int individuo::getValorFuncObj() {
    return valorFuncObj;
}

cromossomo * individuo::getCromo() {
    return cromo;
}

bool individuo::IsEmpty() {
    return(this->cromo->IsEmpty());
}

void individuo::setGeneCromossomo(int Index, int valor) {
    this->cromo->setGene(Index, valor);
}

void individuo::setCromossomo(cromossomo *cromo) {
    this->cromo = cromo;
```

```
    for(int i = 0; i<TAMCROMO; i++)
    {
        this->cromo->setGene(i, cromo->getValorGenes(i));
        this->cromo->getGene(i)->valorToAlelo();
    }
    this->cromo->GenomaFromGenes();
}


int individuo::getValorAleloFromIndex(int index) {
    return (this->cromo->getGene(index)->getValor());
}

void individuo::calculaNaoLinearidade() {
    int
        matriz1[TAMCROMO]; //conterah o calculo do FWT
    int
        i, j, incremento = 0, maxUD = 0, NL = 0, a, b, result;
    char
        buffer[1];

    // Inicio do calculo da FWT a partir da coluna 0
    // de bits do cromossomo.
    // Os valores dos alelos de cada coluna sao
    // trasnferidos para a matriz1.
    j = NUMLOCUS - 1; // iniciando pelos bits menos significativos.

    do
    {
        for(i = 0; i<TAMCROMO; i++)
        {
            buffer[0] = this->cromo->getGene(i)->getAleloFromIndex(j);
            a = atoi(buffer);
            matriz1[i] = a;
        }
        j--; //decrementando j para na proxima execucao
             // trabalhar com a proxima coluna de bits.
        i = 0;
        incremento = 1;

        // Calculo da FWT propriamente dito para cada linha de
        // alelos dos genes do cromossomo.
        do
        {
            do
            {
                a = matriz1[i];
                b = matriz1[i + incremento];
                result = a + b;
                matriz1[i] = result;
                result = a - b;
                matriz1[i + incremento] = result;
                //i irah variar de acordo com o incremento:
                //1, 2, 4, 8 e 16, quanto sairah do laco.
                if((i+1)%incremento == 0)
                {
                    i ++;
                    i += incremento;
                }
                else
                    i++;
```

```
            }while(i < TAMCROMO); //ateh atingir TAMCROMO -1 .

            i = 0;
            incremento <<= 1;
        }while(incremento!=TAMCROMO);

        for(int k = 1; k<TAMCROMO; k++)
        {   //encontra o maxUD para calculo da nao linearidade.
            //maxUD eh o maior valor conseguido na FWT dos bits
            //que compoe os genes do cromossomo.
            if(abs(matriz1[k]) > maxUD)
                maxUD = abs(matriz1[k]);
        }

    }while( j != -1);

    naoLinearidade = (((TAMCROMO - 1) >> 1) + 1) - maxUD;

    if(naoLinearidade ==3)
        printf("XXXXXXXXXX");
    if((maxUD == 5)||(maxUD == 7))
        printf("XXXXXXXXXX");
}

/* Criar o cromossomo aleaoriamente.*/

void individuo::criaCromoAleatorio() {
    cromo->criaGenesAleatorios();
    this->calculaNaoLinearidade();
}

void individuo::downloadIndiv(FILE *arquivo) {
    int valor = 0;
    for(int j = 0 ; j<TAMCROMO; j++)
    {
        valor = this->cromo->getGene(j)->getValor();

        fprintf(arquivo,"%d,", valor);
    }
    fprintf(arquivo,"\n");
}
```

B.2.4 Programa lista.cpp

```
// lista.cpp: implementation of the lista class.

#include "stdafx.h"
#include <stdlib.h>
#include "lista.h"
#include "individuo.h"

// Construction/Destruction

lista::lista() { }

lista::~lista() {
    if(indiv != NULL)
```

```
        delete this->indiv;

//  delete this->next;
}

void lista::setNext(lista *next) {
    this->next = next;
}

lista* lista::getNext() {
    return next;
}

individuo* lista::getIndividuo() {
    return indiv;
}

void lista::setIndividuo(individuo* indiv) {
    this->indiv = indiv;
}
```

B.2.5 Programa newag.cpp

```
// newAg.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include "individuo.h"
#include "populacao.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define TAMPOOL     20
#define TAMPOPINI   5


void geraPool(populacao &pop0, populacao &pool);

void completaPop(populacao &pop);

int main(int argc, char* argv[]) {
  int
    geracao = 0, argNumGeracao = 1, b = -1, i = 0, tipoCorte = -1, controle = 0,
    p1 = 0, p2 = 0;
  populacao
    *newPop, *popIni, *poolIni;

    char
    /*  saida[12],*/ saidaDef[10] = "saida.txt";


 if((argc <= 2)||(argc <= 4))
 {
  printf("\n [ <n> Informar o numero de geracoes.....................................]");
  printf("\n [ < > Informar ou o Tipo de Corte {0 - ALEATORIO, 1 - FIXO(1/2)........]");
  printf("\n [.................................2 - FIXO(1/4), 3 - FIXO(3/4)}.......]");
  printf("\n [ <p> Ou 'p' e dois pontos de corte.....................................]");
```

```
 printf("\n [ < > arquivo onde deve estar a populacao inicial.....................]");
 printf("\n [ < > arquivo onde estarao as populacoes geradas, default eh Saida.txt ]\n\n");
 return 0;
}
 if(strcmp(argv[1],"n") == 0)
  {
   if(argv[2][0] != ' ')
      argNumGeracao = atoi(argv[2]);
   else
   {
    printf("\n [ <n> Informar o numero de geracoes.................................]");
    printf("\n [ < > Informar ou o Tipo de Corte {0 - ALEATORIO, 1 - FIXO(1/2).....]");
    printf("\n [.............................2 - FIXO(1/4), 3 - FIXO(3/4)}.......]");
    printf("\n [ <p> Ou 'p' e dois pontos de corte.................................]");
    printf("\n [ < > arquivo onde deve estar a populacao inicial.....................]");
    printf("\n [ < > arquivo onde estarao as populacoes geradas, default eh Saida.txt ]\n\n");
    return 0;
   }
 }

 if(strcmp(argv[3],"p") == 0)
 {
   if(argv[4][0] != ' ')
   {
     p1 = atoi(argv[4]);
     p2 = atoi(argv[5]);
     if(p1 > p2)
     {
       printf("\n   [ <P> ERRO NA INFORMACAO DOS POSTOS DE CORTE (P1 < P2)]\n");
       return 0;
     }/* Se  P1 == P2 , entao corte aleatorio de tamanho igual a P1. Tratamento
       serah feito na classo populacao.*/

       b = 1;
    }
     else
     {
      printf("\n [ <n> Informar o numero de geracoes.................................]");
      printf("\n [ < > Informar ou o Tipo de Corte {0 - ALEATORIO, 1 - FIXO(1/2).....]");
      printf("\n [.............................2 - FIXO(1/4), 3 - FIXO(3/4)}.......]");
      printf("\n [ <p> Ou 'p' e dois pontos de corte.................................]");
      printf("\n [ < > arquivo onde deve estar a populacao inicial.....................]");
      printf("\n [ < > arquivo onde estarao as populacoes geradas, default eh Saida.txt ]\n\n");
      return 0;
     }
   }
   else
   {
       tipoCorte  = atoi(argv[3]);
       b = 1;
   }

   if(b != -1)
   {

       popIni = new populacao();

       //carrega a populacao inicial
       if(strcmp(argv[3],"p") == 0)
       {
           popIni->loadPovo( argv[6] );//*/"caixas.txt");
```

```
        }
        else
        {
            popIni->loadPovo( argv[4] );//*/"caixas.txt");
        }

        popIni->setGeracao(geracao);

        printf("\nPopulacao %d",popIni->getGeracao());

        /* NEW ... usando POOL. */
        poolIni = new populacao();
        poolIni->setGeracao(geracao);

        geraPool(*popIni, *poolIni);

        /*exclui Populacao Inicial.*/
        delete popIni;


        poolIni->downloadPovo("pools.txt");
        poolIni->downloadPovo("saida.txt");
        poolIni->download("bigPool.txt");
        poolIni->download(); // Download para o povo0.txt.

        delete poolIni;

        do
        {
            // Constroe nova populacao baseada na anterior,
            // ja seleciona as maiores NLs
            printf("\nIniciando Nova Populacao %d", geracao);
            newPop = new populacao(geracao, tipoCorte, p1, p2);

            geracao ++;

            printf("\nFinalizando Nova Populacao %d", geracao);

            delete newPop;

        }while(geracao < argNumGeracao);
        //argNumGeracao informado na linha de comando ou default = 1
    }

    return 0;
}


void geraPool(populacao &pop0, populacao &pool) {
    int maxNL, tamPop;

    maxNL = pop0.getmaxNL();
    tamPop = pop0.getTamanhoPopulacao();

    for(int i = 0; i < tamPop; i++)
    {
        if(pop0.getIndividuo(i)->getNaoLinearidade() >= maxNL)
        {
            pool.setPovo(pop0.getIndividuo(i));
            printf("\nGerando Pool0",pool.getTamanhoPopulacao());
        }
```

```
    }

    if(pool.getTamanhoPopulacao()< TAMPOOL)
    {
        completaPop(pool);
    }
    /* Mantem estatisticas atualizadas.*/
    pool.calcMediasNL();
}

void completaPop(populacao &pop) {
    individuo *aux;

    do
    {
        aux = new individuo();
        aux->criaCromoAleatorio();

        if(((pop.comparaIndividuos(aux))==-1)&&(aux->getNaoLinearidade()>=100))
        {
            pop.setPovo(aux);
            printf("\nCompletando Pool0",pop.getTamanhoPopulacao());
        }
        delete aux;

    }while(pop.getTamanhoPopulacao() < TAMPOOL);

}
```

B.2.6 Programa populacao.cpp

```
#include "stdafx.h"
#include "gene.h"
#include "cromossomo.h"
#include "individuo.h"
#include "populacao.h"

#include "listaMng.h"

#include <stdlib.h>
#include <string.h>
#include <time.h>

#define TAMPOOL 20

populacao::populacao() {
    /*geracao aleatoria da populacao */
    listaPovo = new listaMng();
    this->maxNL = 0;
    this->minNL = 999;
    this->accMaxNL = 0;
    this->accMinNL = 0;
    this->tamanhoPopulacao = 0;
}

populacao::populacao(int geracaoAnt, int tipoCorte, int p1 , int
p2) //populacao
```

```
&popAnterior, {
    int
        numeroAleatorio, cont = 0, contTentativas1 = 0, contTentativas2 = 0, i = 0,
        x = 0;
    individuo
        casal[2], *aux, *auxTmp, *auxTmp1;
    FILE *popAnt1, *popAnt2, *thisPop, *bigPool;

    char buffer[4], buf1[5] = "povo";

    bool testeSemelhante = true;

    listaPovo = new listaMng();


    itoa(geracaoAnt, buffer, 10);
    strcat(buf1, buffer);
    strcat(buf1, ".txt");

    do
    {
        popAnt1 = fopen(buf1, "r");
    }while(popAnt1 == NULL);

    // Linha abaixo, um espelho do arquivo da popAnterior.
    popAnt2 = fopen(buf1, "r");

    strcpy(buf1, "povo");
    itoa(geracaoAnt+1, buffer, 10);
    strcat(buf1, buffer);
    strcat(buf1, ".txt");

    thisPop = fopen(buf1 , "w+");
    if(thisPop == NULL)
    {
        printf("\nERRO ABERTURA ARQUIVO POPULACAO ATUAL. ");
        return;
    }

    this->tamanhoPopulacao = 0;
    this->indicePopulacao  = 0;
    this->geracao = geracaoAnt+1;

    this->maxNL = 0;
    this->minNL = 999;
    this->accMaxNL = 0;
    this->accMinNL = 0;
    this->mediaNL_2_4 = 0;
    /*Tipo de corte do Xover.*/
    this->tipoXcorte = tipoCorte;

    /* Geracao de arquivo temporario para armazenar a populacao
     *  que estah sendo construida pela reproducao
     */
    this->relCabecalho();

    auxTmp1 = (individuo*) 1;

    while((!feof(popAnt1))&&(auxTmp1!=NULL))
    {
```

```
cont = 0;
/* O casal[0] irah ser reproduzido com todos os outros elementos da populacao*/
auxTmp1 = this->leIndivPopAnt(popAnt1);
if(auxTmp1 != NULL)
{
    casal[0].setCromossomo(auxTmp1->getCromo());
    delete auxTmp1;
}

printf("\n Inicio reproducao Populacao %d",this->geracao);


while((!feof(popAnt2))&&(auxTmp1!=NULL))
{
    /* Todos com todos no cruzamento, caso o casal seja composto de 2 individuos
     *  iguais, o segundo eh descartado e incrementado
     */

    auxTmp = this->leIndivPopAnt(popAnt2);

    if(auxTmp != NULL)
    {
        casal[1].setCromossomo(auxTmp->getCromo());
        delete auxTmp;
    }
    // se o casal for igual, le novo individuo.
    if(casal[0].getCromo()->comparaGenes(casal[1].getCromo()))
    {
        auxTmp = this->leIndivPopAnt(popAnt2);
        if(auxTmp != NULL)
        {
            casal[1].setCromossomo(auxTmp->getCromo());
            delete auxTmp;
        }
    }

    if((!feof(popAnt2))&&(auxTmp!=NULL))
    {

        printf("\n Reproducao Populacao %d, i = %d, cont = %d",this->geracao, i, cont);

        /*
         * X-over com corte em um ponto, aleatorio para cada reproducao
         * este cruzamento originara apenas 1 descendente, espera-se assim aumentar a
         * diversidade.
        */
        do
        {
            /* aux serah o filho das reproducoes seguintes.
                Serah deletado tao logo gravado em disco e recriado para
                o proximo cruzamento.*/
            aux = new individuo();

            if(p1 < p2)
            {
                /* X irah variar ateh o ponto 1 de corte. Estes genes serao herdados do
                    Pai[1].*/
                for(x = 0; x < p1; x++)
                {
                    aux->getCromo()->setLocusFromGenoma(x, casal[1].getCromo()->getLocusFromGenoma(x));
                }
```

```
    /* O Numero aleatorio serah igual ao Ponto 2 de corte, verificar o segundo
        laco abaixo para entendimento.*/
    numeroAleatorio = p2;

}
else
{
    /*  P1 e P2 serao iguais e representarao o numero de genes a serem trocados
        durante a reproducao, de forma aleatoria.*/
    if((p1 == p2)&&(tipoCorte == -1))
    {
        numeroAleatorio = pontoXover();
        /* Garantindo que o tamanho do cromo nao serah ultrapassado.*/
        if((numeroAleatorio + p1) > TAMFULL)
            numeroAleatorio = TAMFULL - p1;

        if(numeroAleatorio != 0)
        {
            for(x = 0; x < numeroAleatorio; x++)
            {
                aux->getCromo()->setLocusFromGenoma(x, casal[1].getCromo()->getLocusFromGenoma(x));
            }
        }
        else
            x = 0;

        numeroAleatorio = numeroAleatorio + p1;

    }
    else
    {
        switch(tipoCorte)
        {
        case 0 : numeroAleatorio = pontoXover() ;  break;
        case 1 : numeroAleatorio = TAMCROMO/2   ;  break;
        case 2 : numeroAleatorio = TAMCROMO/4   ;  break;
        case 3 : numeroAleatorio = TAMCROMO*3/4 ;  break;
        default: numeroAleatorio = pontoXover() ;  break;
        }
        /* X eh igual a zero para garantir a copia dos locus desde o inicio do cromo Pai[0].*/
        x = 0;
    }
}


for(x; x < numeroAleatorio; x++)
{
    aux->getCromo()->setLocusFromGenoma(x, casal[0].getCromo()->getLocusFromGenoma(x));
}
for(x; x < TAMFULL; x++)
{
    aux->getCromo()->setLocusFromGenoma(x, casal[1].getCromo()->getLocusFromGenoma(x));
}
/* atribuicao dos novos individuos */
/* Falta avalia-los (povo[i] e povo[i+1])- metodos para isto irah aqui */
aux->getCromo()->GenesFromGenoma();

/* Atribui valor decimal para os genes dos novos individuos */
aux->getCromo()->valorizaGenes();

/* Mutacao para os individuos que apresentam genes repetidos */
```

```
            aux->getCromo()->mutacao();

            aux->getCromo()->GenesFromGenoma();

            aux->getCromo()->valorizaGenes();

            aux->setAptidao(-1);

            aux->calculaNaoLinearidade();

            /* se 1 dos individuos gerados for semelhante a algum da pop inicial, repete o
             * cruzamento - aux.
             */
            if(aux->getNaoLinearidade()< 100)
            {
                this->accMinNL++;
            }
            if(aux->getNaoLinearidade() < this->minNL)
                this->minNL = aux->getNaoLinearidade();

            tamanhoPopulacao = (tamanhoPopulacao) +1;
            indicePopulacao = (indicePopulacao) + 1;

            testeSemelhante = this->comparaPool(aux);
            if(testeSemelhante)
            {
                /* se for diferente da populacao inicial, verifica se existem semelhantes
                 * na nova populacao que estah sendo gerada.
                 */
                if(aux->getNaoLinearidade() > this->maxNL)
                    this->maxNL = aux->getNaoLinearidade();

                if(aux->getNaoLinearidade()>= 100)
                {
                    this->accMaxNL++;
                    aux->downloadIndiv(thisPop);
                    this->relIndividuos(indicePopulacao, aux);

                    bigPool = fopen("bigPool.txt", "a");
                    aux->downloadIndiv(bigPool);
                    fclose(bigPool);
                }

                delete aux;

                cont++;
            }
            else
            {
                printf("\n Individuo igual a um existente.(1) ");
                contTentativas1++;
                delete aux;
            }
        }while((!(testeSemelhante))&&(contTentativas1<1000));
    } //if((!feof(popAnt2))&&(auxTmp!=NULL))

} //fim do while da popAnt1;
// Retorna arquivo para inicio para recomecar os cruzamentos.
rewind(popAnt2);
// Contador para display ao usuario.
i ++;
```

```
    }//fim do while da popAnt2.

    fclose(popAnt1);
    fclose(popAnt2);
    fclose(thisPop);

    /*Estatisticas.*/
    this->mediaNL_4 = ((this->accMaxNL)*1.0/(this->tamanhoPopulacao)*1.0)*100.0;
    this->mediaNL_2 = ((this->accMinNL)*1.0/(this->tamanhoPopulacao)*1.0)*100.0;
    this->relEstatisticas();

}

populacao::~populacao() {
    delete listaPovo;
}

void populacao::setGeracao(int valor) {
    this->geracao = valor;
}

int populacao::getGeracao() {
    return geracao;
}

bool populacao::loadPovo(char *entrada) //leitura do arquivo com
as matrizes. {
    char
        buff[1], string[3] ;

    int
        valorGene, i = 0, indiceGen = 0, indiceIndiv = 0; // indice para indexar atribuicao dos valores dos genes

    individuo
        *aux;


    // Atribui valor inicial para as variaveis.
    for (int x = 0;x<3;x++)
    {
        string[x] = ' ';
    }

    arquivo = fopen(entrada, "r");
    if(arquivo == NULL)
    {
        printf("\n ERRO ABERTURA ARQUIVO %s .\n", entrada);
        return false;
    }
    fread(buff, 1, 1, arquivo);

    aux = new individuo();

    while(!feof(arquivo))
    {
        while((buff[0]!= ' ')&&(buff[0]!= ',')&&(buff[0]!='\n')&&(buff[0]!='\r')&&(!feof(arquivo))&&(i<3)) //
        {
            string[i] = buff[0];
            fread(buff, 1, 1, arquivo);
            i++;
        }
```

```
    if ((string[0] != ' ')&&(string[0] != ','))
    {
        valorGene = atoi(string);
        /* Atribui valores para os individuos que compoe a populacao */
        aux->setGeneCromossomo(indiceGen, valorGene); //this->povo[indiceIndiv]->setGeneCromossomo(indiceGen, valorGene);
        aux->getCromo()->getGene(indiceGen)->valorToAlelo(); //this->povo[indiceIndiv]->getCromo()->getGene(indiceGen)->valorToAlelo();
        //atualiza indice do gene
        indiceGen ++;

        if((indiceGen == 256)) //buff[0] == '\n'
        {
            /* Cria o genoma do individuo atribuindo-lhe todos os valores dos genes de seu cromossomo.
             * O genoma serah usado na reproducao (Xover)
             */
            aux->getCromo()->GenomaFromGenes();//this->povo[indiceIndiv]->getCromo()->GenomaFromGenes();
            aux->setAptidao(15); //this->povo[indiceIndiv]->setAptidao(15);
            aux->setValorFuncObj(0); //this->povo[indiceIndiv]->setValorFuncObj(0);
            aux->calculaNaoLinearidade();

            this->listaPovo->incluir(aux);
            // Atribui novo valor para indice dos individuos.
            indiceIndiv++;
            indiceGen = 0;
            aux = new individuo();
        }
        if((buff[0] == '\r'))
        {
            indiceIndiv++;
            indiceGen = 0;
        }
        // Limpa os valores da string.
        for (int x = 0;x<3;x++)
        {
            string[x] = ' ';
        }
    }
    fread(buff, 1, 1, arquivo); // novo valor para buff de leitura
    i=0;  //zera variaveis utilizadas para leitura do arquivo.
}
/* Cria o genoma do individuo atribuindo-lhe todos os valores dos genes de seu cromossomo.
 * O genoma serah usado na reproducao (Xover) - prevendo o ultimo elemento lido do arquivo, pois os lacos terminam
 * quando encontra-se o fim do arquivo.
 */

if(aux->getCromo()->getGene(0)->getValor() != -1)
{
    aux->getCromo()->GenomaFromGenes();//this->povo[indiceIndiv]->getCromo()->GenomaFromGenes();
    aux->setAptidao(15); //this->povo[indiceIndiv]->setAptidao(15);
    aux->calculaNaoLinearidade();
    aux->setValorFuncObj(0); //this->povo[indiceIndiv]->setValorFuncObj(0);
    this->listaPovo->incluir(aux);
    /* Tamanho da Populacao */
    indiceIndiv++;
}

/* Calculo das estatisticas da nova populacao */
this->tamanhoPopulacao = indiceIndiv;
this->listaPovo->setQtde(indiceIndiv);
this->calcMediasNL();

fclose(arquivo);
```

```
    return true;
}

bool populacao::downloadPovo(char *saida) //todos para arquivo. {
    int valor;


    arquivo = fopen(saida, "a");
    if(arquivo == NULL)
    {
        printf("\n ERRO ABERTURA ARQUIVO %s .\n", saida);
        return false;
    }

    fprintf(arquivo,"\nGeracao Numero: %d \n", this->geracao);
    fprintf(arquivo,"  N.     CROMO                                                                                NLIN\n");
    for(int i = 0; i < (this->tamanhoPopulacao); i++)
    {
        if(i >=1000)
            fprintf(arquivo,"  %d  ", i);
        else
            if(i >=100)
                fprintf(arquivo,"  %d   ", i);
            else
                if(i >=10)
                    fprintf(arquivo,"  %d   ", i);
                else
                    fprintf(arquivo,"  %d     ", i);


        for(int j = 0 ; j<TAMCROMO; j++)
        {
            valor = this->listaPovo->getIndividuo(i)->getCromo()->getGene(j)->getValor();

            /* A cada 16 numeros impresso, pula uma linha e imprime uma tabulacao.*/
            if(((j%16)==0)&& (j!=0))
            {
                fprintf(arquivo,"\n");
                fprintf(arquivo,"\t");
            }

            if(valor >=100)
                fprintf(arquivo," %d ", valor);
            else
            {
                if(valor >=10)
                    fprintf(arquivo,"  %d ", valor);
                else
                    fprintf(arquivo,"   %d ", valor);
            }
        }
        fprintf(arquivo,"   => %d",this->listaPovo->getIndividuo(i)->getNaoLinearidade());
        fprintf(arquivo,"\n");
    }
    /* Imprime as estatisticas */
    fprintf(arquivo,"\n [Tamanho da Populacao..................] = %d",this->tamanhoPopulacao);
    fprintf(arquivo,"\n [Maxima Nao Linearidade Obtida.........] = %d",this->maxNL);
    fprintf(arquivo,"\n [Minima Nao Linearidade Obtida.........] = %d",this->minNL);
    fprintf(arquivo,"\n [Media da Nao Linearidade NL>=100........] = %.2f",this->getmediaNL_4());
    fprintf(arquivo,"\n [Media da Nao Linearidade NL<100........] = %.2f",this->getmediaNL_2());
```

```
    fclose(arquivo);
    return true;
}

bool populacao::operator==(populacao &pop) {
    return (*this == pop);
}

populacao& populacao::operator=(populacao &pop) {
    if(this == &pop)
        return *this;

    for(int i=0;i<(this->tamanhoPopulacao);i++)
    {
        this->listaPovo->incluir(pop.getLista()->getIndividuo(i));
    }
    return *this;
}

void populacao::setPovo(individuo *newIndividuo) {
    individuo *aux;

    aux = new individuo();
    for(int i = 0 ; i < TAMCROMO; i++)
    {
        aux->setGeneCromossomo(i, newIndividuo->getCromo()->getValorGenes(i));
        aux->getCromo()->getGene(i)->valorToAlelo();
    }

    aux->getCromo()->GenomaFromGenes();
    aux->calculaNaoLinearidade();
    this->listaPovo->incluir(aux);
    this->tamanhoPopulacao = this->listaPovo->getQtde();
}

bool populacao::IsEmpty() {
    for(int i=0;i<TAMPOP;i++)
    {
        if(!(this->listaPovo->getIndividuo(i)->IsEmpty()))
            return false;
    }
    return true;
}


int populacao::comparaIndividuos(individuo* pessoa) {
    for(int i =0; i< (this->tamanhoPopulacao); i++)
    {
        if((this->listaPovo->getIndividuo(i)->getCromo()->comparaGenes(pessoa->getCromo())))
            return i;
    }

    return -1;
}


/* Rotinas de Reproducao */

populacao* populacao::reproduz( ) {
    return this;
}
```

```
individuo* populacao::getIndividuo(int index) {
    return (this->listaPovo->getIndividuo(index));
}

int populacao::pontoXover(int limiteSup) {
    time_t t;
    int valor;

    srand((unsigned) time(&t));

    valor = rand()%limiteSup;
    /* Limitando para que o Xover ocorra com pelo menos 2 bits */

    return(valor);
}

void populacao::setTamanhoPopulacao(int tamPop) {
    this->tamanhoPopulacao = tamPop;
}

int populacao::getTamanhoPopulacao() {
    return (this->tamanhoPopulacao);
}

listaMng* populacao::getLista() {
    return(this->listaPovo);
}

bool populacao::procuraIndividuos(individuo* pessoa, int indice) {
    for(int i = 0; i< (this->listaPovo->getQtde()); i++)
    {
       if(this->listaPovo->getIndividuo(i)->getCromo()->comparaGenes(pessoa->getCromo()))
       return true;
    }

    return false;
}

/* Estatisticas*/ void populacao::calcMediasNL() {
    int
        acc2 = 0, acc2_4 = 0, acc4 = 0;

    minNL = 999;
    maxNL = 0;

    for(int i = 0 ; i < this->listaPovo->getQtde();i++)
    {
        if(this->listaPovo->getIndividuo(i)->getNaoLinearidade()>=100)
            acc4++;
        if((this->listaPovo->getIndividuo(i)->getNaoLinearidade()<102)&&(this->listaPovo->getIndividuo(i)->getNaoLinearidade()>100))
            acc2_4++;
        if(this->listaPovo->getIndividuo(i)->getNaoLinearidade()<100)
            acc2++;

        if(listaPovo->getIndividuo(i)->getNaoLinearidade() > maxNL)
            maxNL = listaPovo->getIndividuo(i)->getNaoLinearidade();
        if(listaPovo->getIndividuo(i)->getNaoLinearidade() < minNL)
            minNL = listaPovo->getIndividuo(i)->getNaoLinearidade();
    }
        mediaNL_2   = ((acc2*100.0)/(this->listaPovo->getQtde())*1.0);
```

```
        mediaNL_2_4 = ((acc2_4*100.0)/(this->listaPovo->getQtde()))*1.0);
        mediaNL_4   = ((acc4*100.0)/(this->listaPovo->getQtde()))*1.0);

    for(i = 0 ; i < this->listaPovo->getQtde();i++)
    {
        if(listaPovo->getIndividuo(i)->getNaoLinearidade() == maxNL)
            accMaxNL++;
        if(listaPovo->getIndividuo(i)->getNaoLinearidade() == minNL)
            accMinNL++;
    }
}

double populacao::getmediaNL_2() {
    return mediaNL_2;
}

double populacao::getmediaNL_2_4() {
    return mediaNL_2_4;
}

double populacao::getmediaNL_4() {
    return mediaNL_4;
} int populacao::getmaxNL() {
    return maxNL;
}

void populacao::relCabecalho() {
    FILE *arquivo;

    arquivo = fopen("saida.txt", "a");

    fprintf(arquivo,"\nGeracao Numero: %d \n", this->geracao);
    fprintf(arquivo,"  N.     CROMO                                                                                NLIN\n");

    fclose(arquivo);

}

void populacao::relIndividuos(int indiceindiv, individuo * indiv)
{
    FILE *arquivo;
    int valor;

    arquivo = fopen("saida.txt", "a");

    if(indiceindiv >=1000)
        fprintf(arquivo,"  %d  ", indiceindiv);
    else
        if(indiceindiv >=100)
            fprintf(arquivo,"  %d   ", indiceindiv);
        else
            if(indiceindiv >=10)
                fprintf(arquivo,"  %d    ", indiceindiv);
            else
                fprintf(arquivo,"  %d     ", indiceindiv);


        for(int j = 0 ; j<TAMCROMO; j++)
        {
            valor = indiv->getCromo()->getGene(j)->getValor();
            /* A cada 16 numeros impresso, pula uma linha e imprime uma tabulacao.*/
```

```
            if(((j%16)==0)&& (j!=0))
            {
                fprintf(arquivo,"\n");
                fprintf(arquivo,"\t");
            }

            if(valor >=100)
                fprintf(arquivo," %d ", valor);
            else
            {
                if(valor >=10)
                    fprintf(arquivo,"  %d ", valor);
                else
                    fprintf(arquivo,"   %d ", valor);
            }
        }
        fprintf(arquivo,"   => %d",indiv->getNaoLinearidade());
        fprintf(arquivo,"\n");

    fclose(arquivo);

}

void populacao::relEstatisticas() {
    arquivo = fopen("saida.txt", "a");

    fprintf(arquivo,"\n [Tamanho da Populacao..................] = %d",this->tamanhoPopulacao);
    fprintf(arquivo,"\n [Maxima Nao Linearidade Obtida.........] = %d",this->maxNL);
    fprintf(arquivo,"\n [Minima Nao Linearidade Obtida.........] = %d",this->minNL);
    fprintf(arquivo,"\n [Media da Nao Linearidade NL>=100........] = %.2f",this->mediaNL_4);
    fprintf(arquivo,"\n [Media da Nao Linearidade NL<100........] = %.2f",this->mediaNL_2);

    fclose(arquivo);
}

bool populacao::download(char *saida) {
    FILE *arquivo;
    int valor;

    char buffer[4], buf1[5] = "povo";

    if(saida != NULL)
    {
        arquivo = fopen(saida, "a");
    }
    else
    {
        itoa(this->geracao, buffer, 10);
        strcat(buf1, buffer);
        strcat(buf1, ".txt");
        arquivo = fopen(buf1, "a");
    }

    if(arquivo == NULL)
    {
        printf("\n ERRO ABERTURA ARQUIVO %s .\n", saida);
        return false;
    }

    for(int i = 0; i < (this->tamanhoPopulacao); i++)
    {
```

```
        for(int j = 0 ; j<TAMCROMO; j++)
        {
            valor = this->listaPovo->getIndividuo(i)->getCromo()->getGene(j)->getValor();
            /* A cada 16 numeros impresso, pula uma linha e imprime uma tabulacao.*/

    fclose(arquivo);
    return true;
}


bool populacao::comparaPool(individuo *indiv) {
    FILE *arquivo;

    char
        buff[1], string[3] ;

    int
        valorGene, i = 0, indiceGen = 0, indiceIndiv = 0; // indice para indexar atribuicao dos valores dos genes

    individuo *aux;

    // Atribui valor inicial para as variaveis.
    for (int x = 0;x<3;x++)
    {
        string[x] = ' ';
    }

    arquivo = fopen("bigPool.txt", "r");

    if(arquivo == NULL)
    {
        printf("\n ERRO ABERTURA ARQUIVO %s .\n", arquivo);
        return false;
    }
    fread(buff, 1, 1, arquivo);

    aux = new individuo();

    while(!feof(arquivo))
    {
        while((buff[0]!= ' ')&&(buff[0] != ',')&&(buff[0]!='\n')&&(buff[0]!='\r')&&(!feof(arquivo))&&(i<3))
        {
            string[i] = buff[0];
            fread(buff, 1, 1, arquivo);
            i++;
        }
        if ((string[0] != ' ')&&(string[0] != ','))
        {
            valorGene = atoi(string);
            /* Atribui valores para os individuos que compoe a populacao */
            aux->setGeneCromossomo(indiceGen, valorGene); //this->povo[indiceIndiv]->setGeneCromossomo(indiceGen, valorGene);
            aux->getCromo()->getGene(indiceGen)->valorToAlelo(); //this->povo[indiceIndiv]->getCromo()->getGene(indiceGen)->valorToAlelo();
            //atualiza indice do gene
            indiceGen ++;

            if((indiceGen==256)) //(buff[0] == '\n')||
            {
                aux->getCromo()->GenomaFromGenes();//this->povo[indiceIndiv]->getCromo()->GenomaFromGenes();
                aux->setAptidao(15); //this->povo[indiceIndiv]->setAptidao(15);
                aux->calculaNaoLinearidade();
                indiceGen = 0;
```

```
                if(aux->getCromo()->comparaGenes(indiv->getCromo()))
                {
                    delete aux;
                    fclose(arquivo);
                    return false;
                }

            }

            // Limpa os valores da string.
            for (int x = 0;x<3;x++)
            {
                string[x] = ' ';
            }
        }
        fread(buff, 1, 1, arquivo); // novo valor para buff de leitura
        i=0;  //zera variaveis utilizadas para leitura do arquivo.
    }

    delete aux;
    fclose(arquivo);
    return true;
}


individuo* populacao::leIndivPopAnt(FILE *arquivo) {
    char
        buff[1], string[3] ;

    int
        valorGene, i = 0, indiceGen = 0, indiceIndiv = 0; // indice para indexar atribuicao dos valores dos genes

    individuo *aux;

    // Atribui valor inicial para as variaveis.
    for (int x = 0;x<3;x++)
    {
        string[x] = ' ';
    }
    aux = new individuo();

    fread(buff, 1, 1, arquivo);
    while(!feof(arquivo))
    {
        while((buff[0]!= ' ')&&(buff[0]!='\n')&&(buff[0]!='\r')&&(!feof(arquivo))&&(i<3)&&((buff[0]!=',')))
        {
            string[i] = buff[0];
            fread(buff, 1, 1, arquivo);
            i++;
        }
        if ((string[0] != ' ')&&((string[0] != ',')))
        {
            valorGene = atoi(string);
            /* Atribui valores para os individuos que compoe a populacao */
            aux->setGeneCromossomo(indiceGen, valorGene); //this->povo[indiceIndiv]->setGeneCromossomo(indiceGen, valorGene);
            aux->getCromo()->getGene(indiceGen)->valorToAlelo(); //this->povo[indiceIndiv]->getCromo()->getGene(indiceGen)->valorToAlelo();
            //atualiza indice do gene
            indiceGen ++;

            if((buff[0] == '\n')||(indiceGen==256))
```

```
            {
                aux->getCromo()->GenomaFromGenes();//this->povo[indiceIndiv]->getCromo()->GenomaFromGenes();
                aux->setAptidao(15); //this->povo[indiceIndiv]->setAptidao(15);
                aux->calculaNaoLinearidade();
                indiceGen = 0;
                return aux;
            }

            // Limpa os valores da string.
            for (int x = 0;x<3;x++)
            {
                string[x] = ' ';
            }
        }
        fread(buff, 1, 1, arquivo); // novo valor para buff de leitura
        i=0;  //zera variaveis utilizadas para leitura do arquivo.
    }

    /* retorna null para arquivo vazio.*/
    return NULL;
}
```

B.2.7 Programa fileMng.cpp

```
// fileMng.cpp: implementation of the fileMng class.

#include "stdafx.h"
#include "fileMng.h"

// Construction/Destruction


fileMng::fileMng() {

}

fileMng::~fileMng() {

}
```

B.2.8 Programa listaMng.cpp

```
// listaMng.cpp: implementation of the listaMng class.

#include "stdafx.h"
#include <stdlib.h>
#include "lista.h"
#include "listaMng.h"
#include "individuo.h"

// Construction/Destruction
```

```
listaMng::listaMng() {
    principal = new lista();
    inicio = principal;
    quantidadeIndiv = 0;
    principal->setNext(NULL);
    principal->setIndividuo(NULL);
}

listaMng::~listaMng() {
    lista *aux;

    aux = inicio->getNext();

    while(aux != NULL)
    {
        delete inicio;
        inicio = aux;
        aux = inicio->getNext();
    }
    delete inicio;
}

void listaMng::incluir(individuo *indiv) {
    aux = new lista();
    principal->setNext(aux);
    aux->setIndividuo(indiv);
    quantidadeIndiv++;
    aux->setNext(NULL);
    principal = aux;
}

individuo* listaMng::getIndividuo(int indice) {
    lista *auxLista;

    auxLista = inicio;
    /* Posiciona lista*/
    for(int i = 1;i<=(indice+1); i++)
        auxLista = auxLista->getNext();

    return(auxLista->getIndividuo());
}

void listaMng::setQtde(int valor) {
    this->quantidadeIndiv = valor;
}

int listaMng::getQtde() {
    return (quantidadeIndiv);
}

/* void listaMng::setindividuo(int index, individuo* indiv) {
    individuo *ind;
    lista *auxLista, *novoNoh;

    auxLista = inicio;
    for(int i = 0;i<indice; i++)
        auxLista = auxLista->getNext();

    novoNoh = new lista();
    novoNoh->setIndividuo(indiv);
} */
```